ITHACA INTERSYSTEMS
PASCAL/Z
PASCAL/BZ
USER'S MANUAL
VERSION 4.0

Revised 11/1/81

ITHACA INTERSYSTEMS

PASCAL/Z AND PASCAL/BZ

USER'S MANUAL

VERSION 4.0

by Jeff Moskow

Revised 11/1/81

COPYRIGHT NOTICE.....1

INTRODUCTION.....2
PASCAL/Z OBJECTIVES.....3
IMPLEMENTATION FEATURES.....4
EXTENSIONS TO THE STANDARD LANGUAGE.....4
PASCAL/Z RESTRICTIONS.....5
SYSTEM REQUIREMENTS.....6
RECEIVING INSPECTION.....7
   Contacting Intersystems.....8
CONTENTS OF THE PASCAL/Z DISTRIBUTION DISKETTE.....9
CONTENTS OF THE PASCAL/Z LIBRARY DISKETTE.....10
JENSEN & WIRTH EXAMPLE PROGRAMS.....12
GETTING STARTED.....13

INTRODUCTION TO PASCAL.....14
   Declarations & Definitions.....15
   Intrinsic Constants & Data Types.....16
   Scalar Data Types.....17
   Structured Data Types.....18
   The Pointer Type.....20
   Constants.....22
   Variables.....23
      Global & Local Variables.....23
      Accessing Variables.....24
   Assignment Statements.....25
   Repeat & While Statements.....26
   For Statements.....26
   Conditional Statements.....27
   Compound Statements.....28
   With Statements.....29
   Procedures & Functions.....30
   Pascal Standard Functions.....31
   Pascal Standard Procedures.....31
   Program Structure & Use of Semi-Colons.....32

THE PASCAL/Z COMPILER.....34
   The Symbol Table.....34
   The Type Table.....34
COMPILER OPTIMIZATIONS.....36
VOCABULARY.....37
PASCAL SPECIFICATIONS & LIMITATIONS.....38
   Specifications.....38
   Limitations.....40
COMPILER OPTIONS.....41
OPTIMIZING PASCAL/Z PROGRAMS FOR SPEED.....43
PASCAL/Z TYPE DECLARATIONS.....44
HOW TO RUN PASCAL/Z.....46
   COMPILE.SUB.....48
INTERPRETING PASCAL/Z LISTINGS & ERROR MESSAGES....50
   Run-Time Errors -- Stack Overflow.....51
PASCAL/Z INPUT & OUTPUT.....53
DIRECT FILE ACCESS.....58
RENAME & ERASE.....59
DEVICE INPUT & OUTPUT.....60
PASCAL/Z EXTENSIONS.....61
   Pascal/Z Constants.....63
   Pascal/Z Functions.....64
   Pascal/Z Strings.....66

Pascal/Z CASE Statement.....69
Separate Compilation.....70
Pascal/Z External Routines.....74
Overlaying.....79
INCLUDE Files.....85
CHAINING.....86
PASCAL/Z POINTERS.....87
PASCAL/Z FLOATING POINT NUMBERS.....89
FORMATTING OUTPUT.....90
ASSEMBLER & LINKER ERRORS.....92
MEMORY USAGE.....93
STACK & HEAP ORGANIZATION.....95
INSTALLING PASCAL/Z PROGRAMS IN ROM.....96

APPENDICES.....97
Appendix One -- Parameter Stack Configurations..97
Appendix Two -- Troubleshooting.....98
Appendix Three -- Fixed Point Package.....101
Appendix Four -- Pascal/Z Users' Group.....108
Appendix Five -- Warranty.....109
Appendix Six -- Error Messages.....110
Appendix Seven -- Pascal/BZ.....114

PASCAL/Z COMMENTS & BUG REPORTS.....121

INDEX

COPYRIGHT NOTICE

Wherever referred to throughout these manuals, CP/M and Z-80 are registered trademarks of Digital Research and Zilog, Inc., respectively.

INTRODUCTION

Pascal was designed in 1971 by Niklaus Wirth to:

> "...make available a language suitable to teach programming as a systematic discipline... (and) to develop implementations of this language which are both reliable and efficient on presently available computers."

The Ithaca InterSystems, Inc. Pascal/Z compiler was designed to compile programs written in the Pascal language into Z-80 macro-assembler code. Its design closely follows that of Jensen and Wirth's Pascal User Manual and Report (Second Edition). Pascal/Z is presently available to run under Digital Research's CP/M operating system.

The Pascal/Z software package includes object code for the Pascal/Z compiler in both 48K and 54K versions, object code for the debugger (InterPEST), the assembler (ASMBLE/Z), and the linker/loader (LINK/Z), and both object code and commented source code for the library routines. Also on the diskettes are several example programs, including the .PAS and .COM files for the example programs in the Jensen & Wirth USER MANUAL AND REPORT, and some utility programs to facilitate using the Pascal/Z package. Documentation includes the Pascal/Z Implementation Manual, the InterPEST Reference Manual, the ASMBLE/Z and LINK/Z manuals, and the Jensen & Wirth USER MANUAL AND REPORT.

PASCAL/Z OBJECTIVES

Pascal/Z is a recursive descent Pascal compiler for the Z-80, and was designed to be useful in a variety of environments. Our design objectives were:

1) To run resident on a Z-80.

2) To generate ROM-able re-entrant code.

3) To create a compiler which could be easily re-hosted for use as a cross-compiler.

4) To write the compiler in Pascal for ease of maintenance and reliability.

5) To write a compiler which could be easily modified to generate code for the new 16 bit processors.

6) To produce code which is efficient and to minimize the amount of threaded code.

7) To add extensions necessary for industrial and scientific programming.

While objectives one through six are straightforward, our seventh objective is much more difficult and is constantly being reconsidered. The extensions which have been added were added only if they allowed the user to easily do something that previously would have been impossible or awkward; or if they greatly increase the readability of Pascal programs. In all cases we have striven to maintain the "spirit" of Pascal.

IMPLEMENTATION FEATURES

* The code generated by Pascal/Z is both ROM-able and re-entrant.

* Dedicated Pascal/Z programs may be as small as a few hundred bytes.

* The compiler generates code optimized for a Z-80 processor and takes advantage of special cases.

* Z-80 macro assembly code is generated and optionally includes Pascal source lines as comments to allow user peephole optimization if desired.

* Pascal/Z supports separate compilation of user programs in order to reduce the time to re-compile, re-assemble and re-link large programs.

* Overlays are supported to permit execution of programs larger than system memory size.

* InterPEST (InterSystems Pascal Error Solving Tool), an interactive symbolic debugger designed specifically to isolate and correct faults in a Pascal/Z program.

EXTENSIONS TO THE STANDARD LANGUAGE

* Separate compilation permitted to speed program development.

* Assembly language interface allows versatile EXTERNAL routines.

* INCLUDE files allow insertion of any file within a Pascal/Z program.

* Overlay capabilities provided.

* A STRING type is provided.

* Direct File Access (random access) is supported.

* An ELSE clause is allowed with the CASE statement.

* Symbolic I/O of enumeration types immensely simplifies interactive programs.

* Integer constants can use the operators +, -, *, DIV (i.e., CONST1 = CONST2 DIV CONST3).

* Functions may now return structured types.

* TIME/SPACE optimization is user selectable when using the CASE statement.

See page 61 for more detail on Pascal/Z extensions.

PASCAL/Z RESTRICTIONS

* GOTO may not leave a block (procedure or function).

* Standard GET/PUT I/O is not implemented.

* Procedural parameters not implemented. A procedure/function cannot be passed as a parameter to another procedure/function.

* Dynamic storage is implemented using NEW, MARK and RELEASE rather than NEW and DISPOSE

* The PAGE routine is not implemented.

See page 40 for more detail on Pascal/Z restrictions.

SYSTEM REQUIREMENTS

Pascal/Z runs under the Digital Research CP/M Operating System. We recommend version 2.2, since earlier versions of CP/M did not implement Random File Access, and thus the Random Access (called Direct File Access in Pascal/Z) capabilities of Pascal/Z would be lost when using any early release of CP/M.

While a compilation is in progress, Pascal/Z requires 48K of RAM for itself (54K for the non-overlaying version), plus additional memory for the operating system - usually 8K in the case of CP/M. However, after a program has been compiled, assembled and linked the actual program module may be considerably less than 1K.

The system must also have at least one disk drive while compiling since all compilations read a Pascal source file from the disk and output the resulting macro code to the disk.

A 64K system with two disk drives is recommended for Pascal program development.

RECEIVING INSPECTION

When your Pascal/Z diskette arrives, inspect both the diskette and the shipping carton immediately for evidence of damage during shipping. If the shipping carton is damaged or water-stained, request the carrier's agent to be present when the carton is opened. If the carrier's agent is not present when the carton is opened, and the contents of the carton are damaged, save the carton and packing material for the agent's inspection. Shipping damages should be immediately reported to the carrier.

We advise that in any case you should save the shipping container for use in returning the module to InterSystems, should it become necessary to do so.

Factory Service

Your Pascal/Z compiler comes with updates for one year from the date of original purchase. These updates are available for a nominal copying charge. Contact InterSystems for details.

Before returning the diskette to InterSystems, for any reason, first obtain a Return Authorization Number from our Sales Department. This may be done by calling us, sending us a TWX, or by writing us. After the return has been authorized, proceed as follows:

1) Include an explanatory letter.

2) Include a listing of the malfunctioning program or operation. If no listing is enclosed we can make no guarantee regarding correction of the error, since it may be impossible for us to duplicate the problem without a listing illustrating the malfunction. YOU CAN EXPECT AN ESSENTIALLY IMMEDIATE RESPONSE IF YOU INCLUDE A LISTING. Please send the smallest program which demonstrates the error so that we may isolate and correct the problem in the shortest possible time. If you are working with standard 8", soft-sectored, single density CP/M compatible diskettes, sending the program on disk in addition to the listing would also be helpful. All materials will be returned to you upon request once the problem has been diagnosed.

3) Include the Return Authorization Number.

4) Pack the above information in a container suitable to the method of shipment.

5) Ship prepaid to InterSystems.

Contacting InterSystems:

The following apply for both correspondence and service.

Ithaca InterSystems Inc.
1650 Hanshaw Road
P.O. Box 91
Ithaca, New York 14850
U.S.A.

Telephone (607) 257-0190
TWX 510 255-4346

In Europe:

Ithaca InterSystems (U.K.) Ltd.
58 Crouch Hall Road
London N8 8HG U.K.

Telephone 01-341-2447
Telex 299568

CONTENTS OF THE PASCAL/Z DISTRIBUTION DISKETTE

Before doing anything with your Pascal/Z diskettes, MAKE A BACKUP COPY OF EACH DISKETTE. Use the backup for all of your programming efforts to eliminate the possibility of destroying the distribution diskette--it is in your best interests NEVER TO WRITE ON THE DISTRIBUTION DISKETTES.

If you purchased Pascal/BZ, the business version of the compiler, instead of or in addition to Pascal/Z, see Appendix Seven for information.

The first side of your Pascal/Z Master Diskette contains the following files:

PASCAL48.COM	A preliminary program which loads PAS248
PAS248	The actual compiler (overlaying version)
PASCAL54.COM	The 54K version of PASCAL.COM
PAS254	The 54K version of PAS2
DECS	Overlay module for the 48K version of the compiler
PFSTAT	Overlay module for the 48K version of the compiler
DEBUG.REL	The interactive symbolic debugger, InterPEST (InterSystems Pascal Error Solving Tool)
HOWTO.RUN	A description of how to run the compiler, assembler and linker
HELLO.PAS	Demonstration program
COMPILE.SUB	A submit macro to compile, assemble, link and run Pascal programs (A description of COMPILE.SUB begins on page 49.)
LIB.REL	The Pascal run-time support library
MAIN.SRC	Definitions and routines to be assembled with the output of the compiler
EMAIN.SRC	Routines to be assembled with compiled external Pascal routines and separately compiled modules
XMAIN.SRC	Definitions and routines to be assembled with the output of the compiler when using the debugger
XEMAIN.SRC	Definitions and routines to be assembled with separately compiled modules when using the debugger
ASMBL.COM	Z-80 Macro assembler
LINK.COM	Relocatable linker/loader
INFO.NEW	Hot information not yet included in the manual

The second side of your Pascal/Z Master diskette contains the following files for the Fixed Point Package (see Appendix Three for details):

FIXED.PAS	A collection of procedures which perform arbitrary precision arithmetic in signed fixed-point decimal
FIXCONST.PAS	Declared constants for the fixed-point package

FIXTYPE.PAS	Declared types for the fixed-point package
FIXVAR.PAS	Declared variables for the fixed-point package
FIXEDEX.PAS	Example for use of fixed point package

Also on this side of the Master Diskette are the following files:

PASOPT	Optimizer for Z-80 source output of compiler
CMAIN.SRC	Commented version of MAIN.SRC
OVLYGEN.COM	Program to be used when overlaying
OVERLAY.SRC	Source for the overlay routine in LIB.REL
UCTRANS.*	Programs to allow UCSD --> CP/M file transfer
PRIMES.*	Demonstration program to generate the primes factors of the numbers from 1 to 1000
FILEIO.PAS	A demonstration of Pascal/Z file I/O
EXTENS.PAS	A demonstration program containing at least one example of each Pascal/Z extension.
EXTENS.LST	Listing file generated by Pascal/Z
EXTENS.SRC	Macro code generated by Pascal/Z
EXTENS.REL	Relocatable object code module generated by ASMBLE/Z
EXTENS.COM	Program module generated by LINK/Z. EXTENS cannot be relinked by the user. It is for demonstration purposes only.
CALL.*	Loads Z-80 registers and transfers control to a specified address
RENERA.SRC	Z-80 source for the RENAME and ERASE routines contained in LIB.REL.
RENDRV.PAS	Sample program which drives RENAME
ERADRV.PAS	Sample program which drives ERASE
EXAMPLE.*	A program to calculate the transient program area available in a CP/M system
PEEK.*	PEEK and POKE for Pascal/Z -- allows the user to examine any location in memory and store into that location

Also on this side are the .PAS files for the Jensen & Wirth example programs on the second side of the Library Diskette, provided with permission from Springer-Verlag New York, the publishers of the USER MANUAL AND REPORT.

CONTENTS OF THE PASCAL/Z LIBRARY DISKETTE

The first side of your Pascal/Z Library Diskette contains the following files:

ABSSQR.SRC	Integer absolute value and square floating point absolute value
ADDSUB.SRC	Does BCD addition and subtraction
ARCTAN.SRC	Arctangent function
BYTIN.SRC	Passes a character from file buffer to A register
BYTOT.SRC	Passes a character from A register to file buffer
CHAIN.SRC	Routine to chain Pascal/Z programs

CHKD.SRC	Range check
CLSOT.SRC	Closes output file
CMPCHK.SRC	Floating point routines to complement an operand and check number for a zero
CONSOL.SRC	Console read and print
CVTFLT.SRC	Convert 16 bit integer to floating point number
CVTSFP.SRC	Convert a string to a floating point number
DEFLT.SRC	Contains default values used in assembling some library modules
DIVD.SRC	Divide routines
DONE2.SRC	Finishes two-operand floating point operations
DSKFIL.SRC	Routines to specify device name and delete files from directory
DYNALL.SRC	Dynamic storage allocation and deallocation
ENTEXT.SRC	Enter and exit routines for procedures and functions
EOFLN.SRC	End of file and end of line routines
ERROR.SRC	Closes output files when run time error is encountered
EXPFCT.SRC	Exponential function
FADDSB.SRC	Floating point add and subtract
FCTMAC.SRC	Contains macros used in assembling some library modules
FILEXT.SRC	Routines shared by reset and rewrite routines
FILNAM.SRC	Passes file name to file control block
FLTIN.SRC	Floating point input
FMULT.SRC	Floating point multiply
FOUT.SRC	Floating point output
FPDIVD.SRC	Floating point divide
FPERR.SRC	Floating point routine returns a zero and sets carry bit
FPINIT.SRC	Contains initializations of values used in assembling some library modules
FPMAC.SRC	Contains macros used in assembling some library modules
FPRLOP.SRC	Floating point relational operators
FPSQR.SRC	Floating point square
FPTEN.SRC	Floating point multiply and divide by 10
FXDCVT.SRC	Converts a floating point number to fixed point format
INDIR.SRC	Indirect load and store
INPT.SRC	Read and Readln
LAST.SRC	Marks the last location of the user program
LO.SRC	Routine for closing output files
LOOK.SRC	Looks one character ahead
MAIN.SRC	Definitions and routines to be assembled with the output of the compiler
MPNORM.SRC	Multiple precision add and subtract, and normalizing a floating point number
MULT.SRC	Multiply routines
NATLOG.SRC	Natural log function
OPFILE.SRC	Opens input and output files
OUTPT.SRC	Write and Writeln
PSTAT.SRC	Trace and extended error message routine
RBLOCK.SRC	Calculates random access block number and size, and loads file buffer

RESET.SRC	Resets a file
REWRIT.SRC	Rewrites a file
ROTATE.SRC	Rotates floating point mantissa one bit right or left
ROUND.SRC	Truncate and round functions used to convert floating point to integer
SAVREG.SRC	Saves HL register and sets up pointers to two sets on stack
SETCON.SRC	Routines to construct a set, take the union and intersection of two sets, and test for membership
SETFTN.SRC	Set relational operators, and difference routines
SINCOS.SRC	Sine and Cosine functions
SQRT.SRC	Square root function
SRELOP.SRC	Structural relational operators
STRFCT.SRC	String length, setlength, append, and index
STRLOP.SRC	String relational operators
TEXT.SRC	Buffers flow of characters to and from console
URELOP.SRC	Unstructured relational operators.

Each of these routines is self documented.

JENSEN & WIRTH EXAMPLE PROGRAMS

On the second side of the Library Diskette are the .COM files for the Jensen & Wirth example programs. (The .PAS files are on the second side of the Master.) Also included are data files for those programs which require input. Look at the file JENWIRTH.DOC on the second side of the Library Diskette for detailed information on these programs.

GETTING STARTED

Before doing anything with your Pascal/Z diskette, make a backup copy of the diskette and use that for all of your programming efforts. It is in your best interests NEVER TO WRITE ON THE DISTRIBUTION DISKETTE.

Throughout this manual, whenever a sample dialogue is given, both the computer and user responses will be shown; the user responses should always end with a carriage return.

Now that you have a backup copy of the distribution diskette, insert it into the currently 'logged in' drive (drive 'A' if the system was just booted). In order to verify your copy of the distribution software you may compile, assemble and link HELLO.PAS. To do this try the following dialogue:

```
A>pascal48 hello
InterSystems Pascal v-4.0
HELLO     1--
0 compilation error(s).

A>asmbl main,hello/rel
PASCAL run-time support interface          ASMBL v-7d


A>link hello /n:hello /e
LINK version 2b

A>
```

If any of the above steps did not work as indicated, refer to the section on TROUBLESHOOTING in Appendix Two.

Now that you have successfully compiled, assembled and linked your first Pascal program you may run it as follows:

```
A>HELLO
```

If the program did not welcome you to the land of Pascal/Z then refer to the section on TROUBLESHOOTING.

Once you have verified your diskette by being formally welcomed, continue reading for a brief introduction to the Pascal language. If you are already familiar with Pascal, you may wish to skip to page 34 for information on the Pascal/Z implementation.

If you are using Pascal/BZ, the business version of Pascal/Z, see Appendix Seven for details on how to use BCD numbers, and for an explanation of the differences between Pascal/Z and Pascal/BZ.

INTRODUCTION TO PASCAL

The Pascal/Z Implementation Manual

The first section of the Pascal/Z Implementation Manual contains a brief description of the Pascal programming language and the facilities available to you; however, the discussion here is by no means complete and if you feel that you need additional material, you are advised to refer to a Pascal programming text. Many books about Pascal have recently appeared; among the better ones are:

An Introduction to Pascal by Rodnay Zaks (Sybex, 1980)

An Introduction to Programming and Problem Solving Using Pascal by G.M. Schneider, S.W. Weingart and D.M. Perlman (Wiley, 1978)

and a slightly more advanced and in-depth book:

Programming in Pascal by Peter Grogono (Addison Wesley, 1978)

The Pascal USER MANUAL AND REPORT shipped with the Pascal/Z compiler is an excellent reference manual and is the "standard" by which this compiler was written; it may, however, be a little heavy for the newcomer to Pascal.

The design of the Pascal language was based on the idea that computer programs have two main parts. The first part of a program is the data (variables and constants) and the second part is the "actions" which act upon that data. The data is described by "declarations" and "definitions" while the actions are described by "statements". The following sections decribe how to declare and define the data and then how to write the statements that act upon the data.

This description does not include a discussion of Variant records or GOTO statements: the former because of its complexity and the latter in an effort to discourage its use. The user is referred to the above mentioned texts for a discussion of these topics.

The second section of this manual describes the Pascal/Z implementation of the language. If you are already familiar with Pascal then branch to page 34 for information on using Pascal/Z. If you purchased Pascal/BZ instead of or in addition to Pascal/Z, read Appendix Seven before going on to the rest of the manual.

DECLARATIONS AND DEFINITIONS

Pascal is a "strongly typed" language. This means that each piece of data (whether it is a variable, constant or a parameter) has an associated data type (i.e., integer, real, character....). In addition, each variable must be declared BEFORE it can be used; this allows the compiler to check all operations to make sure that the operands are compatible and are not being misused. There are a number of intrinsic data types (pre-declared by the Pascal language) and there is a facility by which the programmer can construct his/her own data types if desired.

There are two basic classes of data in Pascal. These are scalar and structured. Examples of scalar data are INTEGER, REAL and CHAR. Examples of structured data are ARRAY, FILE, SET and RECORD.

Pascal allows users to create their own data types through use of the TYPE declaration section. This is a section of the program where it is possible to define a type which is used later in the program.

Types other than the standard pre-declared types may be declared in the TYPE declaration section of a program. The following four sections of this manual describe both the pre-declared types and how to declare your own data types.

INTRINSIC CONSTANTS AND DATA TYPES

Pascal has five pre-declared data types which may be used or redefined by the programmer. These types and their attributes are described below.

INTEGER — A variable of type INTEGER is an integer in the range -MAXINT..MAXINT. Pascal has the pre-defined constant MAXINT which is the maximum allowable INTEGER value for a particular implementation (MAXINT = 32767 in Pascal/Z).

REAL — A variable of type REAL is a floating point number. REAL numbers are 32 bits and have a precision of approximately 6 1/2 digits

CHAR — A variable of type CHAR may be assigned any legal ASCII character.

BOOLEAN — BOOLEAN is actually a pre-declared enumeration type with the definition: BOOLEAN = (FALSE, TRUE).

TEXT — TEXT is a pre-declared FILE OF CHAR and is used for I/O.

There is also the pre-defined value NIL. For more information about NIL, see the sections on THE POINTER TYPE and PASCAL/Z POINTERS.

SCALAR DATA TYPES

One very powerful feature of Pascal is the ability to create a new type which is a subrange of an existing type, for example:

```
TYPE INDEX = 1..10;
```

This is useful both because it forces the programmer to decide in advance how a particular variable will be used, and because it allows the compiler to generate code to insure that the particular variable remains within the specified range (if an assignment outside the specified range is attempted, an error will result)..

A programmer may create his own scalar data type, which is referred to as an enumeration type (so called because all of the possible values are enumerated when the user declares the type).

Example of a TYPE declaration section:

```
TYPE COLOR = ( RED, BLUE, YELLOW, GREEN, ORANGE, VIOLET );
     SMALL = 0..99;
     PRIMARY = RED..YELLOW;
     SECONDARY = GREEN..VIOLET;
     LETTERS = 'A'..'Z';
     ASCII = CHAR;
```

COLOR is an enumerated type; PRIMARY and SECONDARY are subranges of COLOR.

In the following examples, the curly brackets - {} - are the standard Pascal comment markers. In Pascal/Z "(*" and "*)" may be used for the same purpose.

The COLOR definition will be used in examples throughout this manual.

STRUCTURED DATA TYPES

Pascal allows the user to construct his own data type by combining other types. Each of the major structures will be discussed in this section.

First is the ARRAY (similar to the array in BASIC). An ARRAY has two basic parts: the index type and the element type. The index type determines how a particular point in the ARRAY is referenced and the element type defines what type of data exists at that point.

```
{ THIS WILL BE AN ARRAY INDEXED FROM 1 TO 50 AND }
{ EACH ELEMENT IS OF TYPE INTEGER }
TYPE TABLE = ARRAY[ 1..50 ] OF INTEGER;

{ THIS ARRAY IS EQUIVALENT TO THE LAST ONE }
TYPE INDEX = 1..50;
   ELEMENT = INTEGER;
   TABLE = ARRAY[ INDEX ] OF ELEMENT;

{ THIS WILL BE AN ARRAY, INDEXED BY SIZE, OF PRICE }
TYPE SIZE = ( TINY, SMALL, MEDIUM, LARGE, XLARGE );
   PRICE = REAL;
   CHART = ARRAY[ SIZE ] OF PRICE;
```

The next data structure is the RECORD. Like the ARRAY, the RECORD allows the programmer to combine many pieces of data into one type. A RECORD is a collection of elements (which may be of different types) where each element is given a unique name.

```
{ A RECORD TO SPECIFY THE SIZE AND WEIGHT OF AN OBJECT }
TYPE OBJECT = RECORD
                SIZE: ( TINY, SMALL, MEDIUM, LARGE, XLARGE );
                WEIGHT: REAL
              END;

{ A RECORD TO SPECIFY A DATE AND TIME... }
{ ...AND A CALENDAR OF IMPORTANT DATES }
TYPE MONTHS = ( JANUARY, FEBRUARY, MARCH, APRIL, MAY,
                JUNE, JULY, AUGUST, SEPTEMBER, OCTOBER,
                NOVEMBER, DECEMBER );
     DATE = RECORD
              MONTH: MONTHS;
              DAY: 1..31;
              YEAR: 1900..2100;
              HOUR: 1..12;
              MINUTE, SECOND: 0..59
            END;
     CALENDAR = ARRAY[ ( BIRTHDAY, ANNIVERSARY, GRADUATION,
                VACATION, HOLIDAY, PAYDAY ) ] OF DATE;
```

```
{ A RECORD TO RECORD INFORMATION ABOUT A FRIEND }
TYPE NAME = ARRAY[ 1..30 ] OF CHAR;
     ADDRESS = RECORD
                STREET: ARRAY[ 1..30 ] OF CHAR;
                NUMBER: 0..MAXINT;
                TOWN, CITY: ARRAY[ 1..30 ] OF CHAR;
               END;
     FRIEND =  RECORD
                WHO: NAME;
                WHERE: ADDRESS
               END;
```

The next type of structure is a SET. A SET is a collection of objects of a given type. A SET is declared to have a particular element type. Each SET variable may contain none, one, some or all of the possible elements for that SET. For example, consider the following:

```
TYPE STOCK = ( PAINT, BRUSHES, THINNER, TOOLS, ROLLERS );
     IN_STOCK = SET OF STOCK;
```

A variable of type IN_STOCK will indicate for each of the possible values of STOCK, whether or not the particular element is present.

Pascal allows the programmer to test for SET membership, inclusion (is one SET a subset of another), equality and inequality. In addition the programmer may take the difference, union or intersection of two SETs.

Here are some more examples of SET declarations:

```
TYPE DIGITS = SET OF 0..9;
     ATTENDEES = SET OF ( DAVE, BILL, STEVE, JEFF, ROB );
```

The last structured type is the FILE. FILEs are used to communicate with the world outside of Pascal. Each file has a particular element type so that both you and the compiler know what type of data will be read or written. The following examples should help to clarify the use of FILE types.

```
{ A TEACHER'S GRADE RECORDS MIGHT BE STORED AS... }
TYPE BLUE_BOOK = FILE OF RECORD
                   NAME: ARRAY[ 1..30 ] OF CHAR;
                   GRADE: 'A'..'F'
                 END;

{ A LIST OF #'S MIGHT BE STORED AS.... }
TYPE NUMBERS = FILE OF INTEGER;
```

THE POINTER TYPE

The last type of data is a pointer. A pointer variable is one which points to another variable of a specified type. This type is useful when constructing linked lists or other data structures where one datum has a need to know about another. In Pascal pointers are used in connection with dynamic variables -- variables which DO NOT have to be declared ahead of time.

A common example is the creation of a linked list of records. The idea is this: a type is defined as a pointer to a record of type; the record of type is the kind of record we want to have linked together and created dynamically. An example (from Jensen/Wirth User Manual and Report):

```
type link = ^person;
     ...
     person = record
          ...
          next: link;
          ...
     end;
```

Here, person is a type defined as a record presumably with various fields in it concerned with age, health, sex, social security number (these fields are indicated by the periods), but including at least one field which is a pointer to a record of the same type.

After such a type definition, the variable declaration would be something like this:

```
var first, p: link;
```

To write a program that creates a linked list of these records, the Pascal built-in procedure NEW is used. Every time the program statement NEW(p) is encountered, it will create another record of type person, because "p" is a variable of type link which is defined as a pointer to the type person. In addition, the NEW procedure assigns to the variable p the value that points to this newly created record. Since this record that is being created by each NEW(p) statement has, as one of its fields, a pointer variable, one may link each newly created record to the next record by proper arrangement of assignment statements. Many schemes are possible, with pointers that point both forward and backward, but the important thing to remember is that the procedure NEW creates an entirely new variable. The example from the User Manual and Report goes like this (using the types declared above):

```
var p, first: link;
    i: integer;
    socialsecurity: file of array[1..3] of integer;
     ...
begin
     ....
     first := nil; { "nil" is a special value which
                     indicates that the pointer points
                     to nothing }
     for i := 1 to n do
         begin
          read( socialsecurity, s );
          new( p );
          p^.next := first;
          p^.ss := s;
          first := p
         end;
     ....
     end.
```

What happens here is that n records are created, of the type person. "p^.ss := s" assigns a value to a field in the present record, s is retrieved from a file in the previous read statement. "p^.next := first" assigns the value of the pointer "first" to the pointer field that was built into the record; then "first := p" makes first point to the present record.

Each time this group of statements is executed by the FOR loop, the pointer "first" is set to point to the most recently created record, and the pointer field within that record is set to point to the previous record.

To get the data out of such a structure, a program could use the "first" pointer to locate the most recent record, and then the pointer within the record to reference the second record, and so forth. One might create another variable of type pointer, which could be assigned the values of the pointers from each record, one after another, to inspect the other fields in the record (stopping when the "next" pointer is nil).

The advantage of such a data structure is largely its dynamic nature, since it is not necessary to declare the size of the list ahead of time.

Dynamically created storage may be returned when it is no longer needed (see the section on Pascal/Z pointers).

To assist those users with non-standard keyboards, the pointer symbol ^ has the ASCII value 5E hex.

A later section, PASCAL/Z POINTERS, decribes Pascal/Z's implementation of pointers.

CONSTANTS

Many programs make use of constants such as the maximum number of items, size of table, perfect score, etc. In Pascal it is possible to define scalar constants and strings symbolically in the CONST declaration section. In addition to making your program more legible and easier to maintain, the use of constants rather than variables will help the compiler to generate more compact and efficient code.

Example of CONSTant declarations:

```
CONST     TABLE_SIZE = 50;
          VERSION = 'Version 1a';
          PI = 3.1415926;
          NEGATIVE = -TABLE_SIZE;
```

It is clearly advantageous to declare constants and use them throughout your program since it gives you the ability to change the value of all occurrences of the constant throughout the entire program by redefining one constant at the beginning (e.g., changing all references to a table size from 50 to 100).

Pascal defines a constant string, such as VERSION in the example, to be:

```
ARRAY[ 1..N ] OF CHAR
```

where N > 1. If N = 1 then the constant is of type CHAR, not ARRAY OF CHAR.

VARIABLES

All variables must be declared before a Pascal program is allowed to refer to them. In addition each variable must be explicitly declared to be of a certain type. The type of a variable determines the amount of storage that it requires, as well as defining how it may be used throughout the program.

Example of VARiable declarations (the two groups of declarations here are equivalent):

```
VAR  COUNTER: ARRAY[ -4..20 ] OF COLOR;
     FLAG: BOOLEAN;
     SIZE: ( TINY, SMALL, MEDIUM, LARGE, XLARGE );
     I,J: INTEGER;
     TABLE: ARRAY[ 1..50 ] OF INTEGER;
     LETTER: 'A'..'Z';
     TEMPERATURE: REAL;
```

and

```
CONST TABLE_SIZE = 50;

TYPE EXPANSE = ( TINY, SMALL, MEDIUM, LARGE, XLARGE );
     LETTERS = 'A'..'Z';
     NUMBER = INTEGER;
     TABLE_RANGE = 1..TABLE_SIZE;
     TABLE_ELEMENT = INTEGER;

VAR  COUNTER: ARRAY[ -4..20 ] OF COLOR;
     FLAG: BOOLEAN;
     SIZE: EXPANSE;
     I,J: NUMBER;
     TABLE: ARRAY[ TABLE_RANGE ] OF TABLE_ELEMENT;
     LETTER: LETTERS;
     TEMPERATURE: REAL;
```

Global and Local Variables

A global variable is one which is defined under the main program heading. It may be used, referenced or changed anywhere within the program. Global variables may be accessed by any procedure or function which does not use the same identifiers.

A local variable is one which is defined within a procedure or function. It will disappear once the procedure or function has been executed, and will have no effect outside its procedure or function. A local variable will have no effect on a global variable.

Accessing variables

Variables are accessed simply by referring to the variable by name. However, sometimes it is desirable to refer to a particular part of a variable, such as an ARRAY element, a field of a RECORD, or the target of a pointer. These types of accesses are illustrated below:

```
TABLE[ 24 ]          { THE 24TH ELEMENT OF OUR ARRAY }
DATE.MONTH           { THE MONTH FIELD OF A DATE RECORD }
FRIEND.NAME[ 1 ]     { THE FIRST CHARACTER OF A NAME }
FRIEND.NAME[ I ]     { THE Ith CHARACTER OF A NAME }
LINK^                { THE RECORD POINTED TO BY LINK }
LINK^.SIZE           { THE SIZE FIELD IN THIS RECORD }
```

Now that you know about the intrinsic data types, Now that you know about the instrinsic data types, how to create new data types, declare constants and declare variables, the following sections will describe the statements which manipulate the data.

ASSIGNMENT STATEMENT

One of the most basic statements in any language is the assignment statement (called the LET statement in BASIC). All assignment statements in Pascal have the same form, which is:

<variable> := <expression>

Where <expression> may be any combination of constants, variables and function calls, as long as the type of <expression> is compatible with that of <variable>.

Examples of assignment statements are:

```
{ two assignments to a variable of type INTEGER }
A := 4;
RESULT := X * Y DIV Z;

{ assignment to a variable of user defined type COLOR }
STOCK.ITEM := GREEN;

{ assignment to an element of an ARRAY OF CHAR }
TABLE[ 20 ] := 'Z';

{ This is an assignment to a variable of type BOOLEAN }
{ the effect of this statement is to:                 }
{     1) INTEGER divide A by B                        }
{     2) multiply the result of step 1 by B           }
{     3) compare result of step 2 to A                }
{     4) assign result of the comparison to DIVISIBLE }
{ this works without parenthesis because relational   }
{ operators are the lowest precedence operators       }
DIVISIBLE := A DIV B * B = A;

{ The variable MY_NAME is of type:                    }
{     ARRAY[ 1..4 ] OF CHAR                           }
{ This assignment copies one ARRAY into another       }
MY_NAME := 'Lyla';

{ This assignment copies one RECORD into another of     }
{ the same type. (The type of these RECORDs is DATE     }
{ which was defined in the section on structured types) }
PARTY_DATE := BIRTH_DATE;
```

REPEAT AND WHILE STATEMENTS

The REPEAT and WHILE statements are used to execute a loop until some desired condition is reached. The main difference between the two statements is that the REPEAT loop does a test for completion at the end of the loop; and the WHILE loop does a test at the beginning. This means that the body of a REPEAT loop is executed at least once before the loop is terminated, but the same is not true for a WHILE loop. The form of these statements is:

```
REPEAT <one or more statements separated by ; >
     UNTIL <expression>

WHILE <expression> DO <statement>
```

In both statements <expression> must be a BOOLEAN expression. The REPEAT loop terminates when the expression is TRUE and the WHILE loop exits when the expression is FALSE.

FOR STATEMENT

The FOR statement should be used in place of the REPEAT or WHILE statements when it is possible to determine in advance the number of repetitions necesssary. The format of a FOR statement is:

```
FOR <control variable> := <initial value> TO/DOWNTO
                          <final value> DO
```

The type of variable allowed is any variable of scalar type except REAL. The following examples should give an idea of the flexibility of this statement:

```
FOR INDEX := 1 TO 50 DO <statement>

FOR ITEM := PAINT TO ROLLERS DO <statement>

FOR TINT := VIOLET DOWNTO RED DO <statement>
```

CONDITIONAL STATEMENTS

The next statement is the IF statement (similar to the IF statement in BASIC). Its form is:

```
    IF <expression> THEN <statement>
or
    IF <expression> THEN <statement> ELSE <statement>
```

The expression must be a BOOLEAN expression and the statement following the THEN is executed if the expression is TRUE, otherwise the statement following the ELSE (if there is an ELSE) is executed instead.

There is another conditional statement, the CASE statement. The CASE statement is similar to, but more powerful than, the ON...GOTO statement in BASIC. Its form is:

```
    CASE <case selector> OF
     <case label list>: <statement>;
          ......
     <case label list>: <statement>
    end;
```

The <case selector> may be a variable or an expression. Each <case label list> is a list of constants (they MUST be constants) for use in selecting a particular CASE. Here is an example of the use of the CASE statement:

```
    CASE TINT OF
     WHITE, RED: <statement>;
     GREEN: <statement>;
     YELLOW, ORANGE, BLUE: <statement>
     END;
```

COMPOUND STATEMENTS

Sometimes it is desirable to have a group of statements act as one statement so that the programmer can insert the group into a construct which expects only one statement. For example it might be useful to be able to do the following, which is not legal in Pascal:

```
IF <expression> THEN
     <statement 1>
     <statement 2>
          .....
     <statement n>
ELSE <statement>
```

This can easily be done in Pascal by constructing a compound statement. The compound statement takes the form:

```
BEGIN
  <statement>;
  <statement> { This line may be repeated zero or more times}
END
```

The above example becomes:

```
IF <expression> THEN
    BEGIN
     <statement 1> ;
     <statement 2> ;
          .....
     <statement n>
    END
ELSE <statement>
```

A compound statement may be used ANYWHERE a statement is used.

WITH STATEMENT

The WITH statement is used when a particular RECORD is being accessed repeatedly. It is both shorthand for the programmer and in some cases allows the compiler to produce more efficient code. The CALENDAR RECORD from the RECORD description section is used in this example of the WITH statement:

```
VAR C: CALENDAR;
BEGIN
    C[ BIRTHDAY ].MONTH := JANUARY;
    C[ BIRTHDAY ].DAY := 17;
    C[ BIRTHDAY ].YEAR := 1958;
    C[ BIRTHDAY ].HOUR := 12;
    C[ BIRTHDAY ].MINUTE := 18;
    C[ BIRTHDAY ].SECOND := 0;
END;
```

The following group of statements accomplishes the same thing:

```
VAR C: CALENDAR;
BEGIN
    WITH C[ BIRTHDAY ] DO BEGIN
     MONTH := JANUARY;
        DAY := 17;
        YEAR := 1958;
        HOUR := 12;
        MINUTE := 18;
        SECOND := 0;
    END;
END;
```

Notice how the compound statement was used in the second example to reduce the verbiage and increase the readability of the code.

PROCEDURES AND FUNCTIONS

In Pascal there are two types of subroutines -- procedures and functions. Functions are basically the same as procedures except that functions return a scalar value (e.g., ABS, SIN, SQRT, etc.). Procedures are called explicitly as statements (procedure statements). Functions are used in expressions.

Pascal allows parameters to be passed to procedures/functions. There may be as many parameters as desired and each parameter may be passed either by value or by reference (VAR parameter). In addition each procedure/function may have its own local constants, types and variables. Procedures and functions may be nested and each can access the constants, types and variables of all surrounding blocks. All procedures/functions are fully recursive. Listed below are some examples of procedures and functions:

```
TYPE STUDENTS = ( ROB, JEFF, BILL, LAURIE, LYLA, NEAL );
     TEST_SCORES = ARRAY[ STUDENTS ] OF 0..100;
{ A PROCEDURE TO OUTPUT THE SCORES OF A TEST }
PROCEDURE RESULTS( TEST_N: TEST_SCORES );
VAR STUDENT: STUDENTS;
BEGIN
    FOR STUDENT := ROB TO NEAL DO
     BEGIN
         WRITE( STUDENT:10, TEST_N[ STUDENT ]:3 );
         IF TEST_N[ STUDENT ] = 100 THEN WRITELN( 'A+' )
         ELSE WRITELN;
     END;
END;

{ A FUNCTION TO RETURN THE MAXIMUM OF TWO NUMBERS }
FUNCTION MAX( X, Y: INTEGER ): INTEGER;
BEGIN
    MAX := X;
    IF X < Y THEN MAX := Y
END;
```

PASCAL STANDARD FUNCTIONS

Pascal contains a variety of pre-defined functions which are available to the programmer. Each function and its description are listed below:

ABS(X)	Return the absolute value of X. This function takes either a REAL or an INTEGER argument and returns a value of the same type.
SQR(X)	Return the square of X. This function takes either a REAL or an INTEGER argument and returns a value of the same type.
ODD(X)	The type of X must be INTEGER. Returns TRUE if X is odd, FALSE otherwise.
CHR(X)	The type of X must be INTEGER and the result is the character with the ordinal value of X.
ORD(X)	The type of X may be any scalar except REAL and the result is the ordinal number of that element.
PRED(X)	The type of X may be any scalar except REAL and the result is the value which precedes X.
SUCC(X)	The type of X may be any scalar except REAL and the result is the value which succeeds X.
EOLN(X)	The type of X must be FILE OF CHAR (i.e., TEXT) and the result is TRUE if the file is at the end of the current line.
EOF(X)	The type of X must be FILE and the result is TRUE if end-of-file has been reached.
TRUNC(X)	The type of X must be REAL and the result is the INTEGER whose absolute value is less than X.
ROUND(X)	The type of X must be REAL and the result is the INTEGER whose value is closest to X.

The following functions take either a REAL or INTEGER argument and return a REAL result.

SIN(X)	The trigonometric SINE of X.
COS(X)	The trigonometric COSINE of X.
ARCTAN(X)	The trigonometric ARCTANGENT of X.
SQRT(X)	The square root of X.
LN(X)	The Natural Logarithm of X.
EXP(X)	e raised to the Xth power.

PASCAL STANDARD PROCEDURES

Pascal defines a number of standard procedures which may be used by the programmer. They are used in I/O and dynamic storage allocation/deallocation. For a discussion of these procedures, see the sections on PASCAL/Z INPUT AND OUTPUT, THE POINTER TYPE and PASCAL/Z POINTERS.

PROGRAM STRUCTURE AND USE OF SEMI-COLONS

The actual format of Pascal programs is not fixed. You may put symbols anywhere on a line and any statement or declaration may be spread over many lines, as long as each individual symbol is not broken by any separators. However, there is a structure to Pascal programs which must be followed. This structure is:

```
<program> ::= PROGRAM <program identifier> ;
              <block>

<block>   ::= <label declaration part>
              <constant declaration part>
              <type declaration part>
              <variable declaration part>
              <statement part>
```

For a more detailed description of program structure see the syntax tables in Jensen and Wirth, page 110.

One of the more confusing aspects of Pascal is where to put semi-colons. To the Pascal newcomer it seems as if their placement is purely arbitrary. However, there is a rationale behind this scheme. Semi-colons are used to separate two adjacent statements and to terminate a declaration. The following are all legal Pascal program fragments:

```
{ NOTICE HOW EACH DECLARATION IS TERMINATED BY A ';' }
CONST MAX = 100;
      TITLE = 'World''s greatest program';
TYPE  COLOR = ( RED, BLUE, YELLOW, GREEN );
      LETTERS = 'A'..'Z';
VAR   I,J: INTEGER;
      HOUSE: COLOR;

{ This shows adjacent statements separated by ;'s          }
{ BEGIN....END is one statement so there is no need to     }
{ put a semi after BEGIN or before END. Similarly          }
{ REPEAT....UNTIL is one statement so there is no          }
{ need to put a semi after REPEAT or before UNTIL.         }
BEGIN
    I := 4 ;
    HOUSE := RED;
    REPEAT
        HOUSE := SUCC( HOUSE )
    UNTIL HOUSE = GREEN
END
```

```
{ Pascal allows null statements, for example the }
{ following group of statements contains an IF   }
{ statement with no effect because the statement }
{ following the THEN is a null statement         }
BEGIN
    HOUSE := SUCC( HOUSE );
    IF HOUSE = GREEN THEN ;
    I := 23
END

{ Pascal programmers should be careful when using null     }
{ statements since they can introduce subtle bugs into     }
{ Pascal programs. The following two code segments         }
{ illustrate this. If X is less than Y the first           }
{ segment loops for a while and then continues and the     }
{ second segment loops forever                             }
BEGIN
    WHILE X < Y DO
     BEGIN
         WRITE( X );
         X := X + 1
     END
END

and
BEGIN
    WHILE X < Y DO ;
     BEGIN
         WRITE( X );
         X := X + 1
       END
END
```

THE PASCAL/Z COMPILER

The Pascal/Z compiler accepts as input a program written in the Pascal programming language (.PAS file), and outputs two files. One contains the Z-80 macro-code generated by the compiler, and is specified as <filename>.SRC. The other file output is the listing file, or .LST file. This file contains the original Pascal program, with pagination, line numbers, statement numbers, and nesting levels indicated. Any errors in the program are also indicated, and their location in the program specified.

There are four sections to the Pascal/Z compiler. The first is PASCAL48.COM, which contains initialization code for PAS248 -- the actual compiler. PASCAL48 chains to PAS248 after initialization is completed. There are two overlay modules called by PAS248: DECS and PFSTAT. This version of the compiler requires a minimum of 48K to compile -- approximately 37K for the PAS248 code, 8K of symbol and type table and overlay space, and 3K of stack space.

Two versions of the compiler are supplied on the distribution diskette. The first is described above. The difference between it and the second version is that the second version, PASCAL54, runs in 54K. The symbol and type tables are approximately the same size as those for the smaller compiler, but the 54K version will compile about twice as quickly because it does not use overlays. This will have no effect on the execution speed of the final code.

The 54K version which has slightly larger symbol and type tables and which does not use overlays is named PASCAL54.COM and PAS254 on the diskette. It requires 54K to compile. To invoke the 54K compiler, type 'PASCAL54' rather than 'PASCAL48'. PASCAL54.COM chains automatically to PAS254.

The Symbol Table

The symbol table for the 48K version will accept up to 400 entries, and that for the 54K version will accept up to 500 entries. An entry is made into the symbol table for every identifier declared in the Pascal/Z program.

The Type Table

The type table is structured differently from the symbol table in that it is allocated in bytes rather than entries. The type table for the 48K version is 1500 bytes in size, and that of the 54K version is 1700 bytes in size. Any declaration of a type will result in the appropriate number of bytes being allocated in the type table.

There are a number of pre-defined entries in both the symbol and type tables (such as INTEGER, REAL, CHAR, BOOLEAN, TEXT).

All local entries to the symbol and type tables (except for those which are predefined above) are deallocated upon exitting a block so that the space may be reused.

See the section on PASCAL/Z TYPE DECLARATIONS for information on how to reduce type table usage.

PASCAL/BZ

Pascal/BZ is a version of the original Pascal/Z compiler which has been modified to accommodate the business user. The Pascal/Z floating point routines are replaced in Pascal/BZ by BCD (binary-coded decimal) fixed point routines to allow for the greater precision and accuracy necessary for the business programmer.

If you licensed Pascal/BZ in addition to or instead of Pascal/Z, you will find a description of the differences between the two in Appendix Seven of this manual, as well as detailed information on how to use Pascal/BZ BCD numbers.

COMPILER OPTIMIZATIONS

This section contains a list of the kinds of optimizations performed by the compiler.

1) Constant folding.

2) Temporary variable allocation.

3) Expression reordering for optimal register and temporary use.

4) Different code generation for one byte expressions involving single byte values.

5) Processor independent peephole optimization.

6) Increments/decrements used in place of adds/subtracts wherever possible.

7) Additions used in place of multiplications wherever possible.

8) Variable storage is ordered to allow faster variable access.

9) Processor specific optimizations exploited.

10) Compiler always exploits the use of a non-real constant.

11) Whenever possible a check is performed at compile-time (except for divide by zero which is detected and ignored until run-time).

VOCABULARY

All Pascal programs are made up of the following basic symbols:

<basic symbol> ::= <letter> | <digit> | <special symbol>

<letter> ::= a | b | c | d | e | f | g | h | i | j | k | l | m |
n | o | p | q | r | s | t | u | v | w | x | y | z |
A | B | C | D | E | F | G | H | I | J | K | L | M |
N | O | P | Q | R | T | S | U | V | W | X | Y | Z |
$ | # | _

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<special symbol> ::= + | - | * | / | = | <> | < | > | <= | >= |
(|) | (. | .) | [|] | := | . | , | ; |
: | ' | ^ | .. | (* | *) | { | } |
<reserved word>

<reserved word> ::= and | array | begin | case | const | div |
do | downto | else | end | external | file |
for | forward | function | goto | if | in |
label | mod | nil | not | of | or | packed |
procedure | program | record | repeat | set |
string | then | to | type | until | var |
while | with

In Pascal/Z the following also applies:

Only standard ASCII characters may be used in writing Pascal/Z programs. Any other characters may be used only in quoted strings. Introducing non-standard ASCII characters (i.e. control characters) into a Pascal/Z program will cause inconsistent, and often disastrous, results.

The use of some editors when composing a Pascal/Z program may cause parity bits to be set. This can cause strange results, and can be remedied by stripping the parity bits from the file before compiling the program.

(. is equivalent to [
(* is equivalent to {
*) is equivalent to }
$ # _ have been defined as letters
UPPER case letters are equivalent to lower case letters
<space> <comment> <cr> <lf> <ff> <tab> are separators

Separators may not occur within numbers, strings, identifiers or reserved words.

At least one separator is required between any two consecutive numbers, identifiers or reserved words.

PASCAL/Z SPECIFICATIONS AND LIMITATIONS

In many places throughout the PASCAL REPORT, Jensen and Wirth leave certain aspects of Pascal undefined. This section describes what happens in the Pascal/Z implementation. The user is cautioned against writing programs which depend on these since they are implementation specific and are therefore not guaranteed to be transportable.

SPECIFICATIONS

MAXINT:	32767 (The maximum integer allowable under Pascal/Z). The smallest integer allowable is -32767.
REALS:	REAL numbers in Pascal/Z are 32 bits and have a precision of 6-1/2 digits (the 7th digit may contain a round-off error).
LARGEST REAL:	The largest allowable REAL number in Pascal/Z is 1.7E+38. The smallest is 2.9E-39.
MAXIMUM SET SIZE:	256 elements in the range 0 to 255. All sets are 32 bytes in size.
LINE/SYMBOL LENGTH:	The maximum allowable length of any line or symbol is eighty characters.
CONSOLE INPUT:	The buffer will hold up to 80 characters.
STRING LENGTH:	The maximum string length is 255.
IDENTIFIERS:	The first eight characters of all identifiers are significant.
RECORDS:	No RECORD may have more than forty fields. This can be circumvented by nesting the records.
OUTPUT FILES:	The maximum number of open output files is four. This may be changed by redefining MAXOUT in MAIN.SRC. See the section on PASCAL/Z INPUT AND OUTPUT for further detail.
PARAMETERS:	The maximum number of parameters which may be passed to a procedure or function is fifteen.
NESTING LEVELS:	The maximum number of nesting levels is fifteen.

GOTO statement: Jumps out of procedures and functions are not allowed (This is generally a bad practice).

Jumps out of FOR loops will take up four bytes of storage for each loop exitted prematurely (a jump out of a nested FOR loop will result in a disaster).

Jumps into structured statements are not recommended but will work properly as long as the jump is into a REPEAT or WHILE loop.

BOOLEAN EXPRESSIONS: These are only evaluated as far as necessary (i.e., when executing the statement 'IF A OR B THEN...', the variable B is not evaluated if A is TRUE, since the value of the expression is already known).

FUNCTIONS: Non-real scalar functions return the value zero (or FALSE) if no assignment to the function identifier is made. Structured functions have no default value.

CASE statements: The statement after the CASE is executed if none of the case lists is selected (no error).

JUMP TABLE SIZE: The maximum jump table size when using the CASE statement is 76 entries including 0.

WITH statement: WITH A[I] DO BEGIN.... if I changes, as long as the WITH statement is not exitted the original record will be referenced (i.e., the expression is NOT re-evaluated).

PACK/UNPACK: Arrays and records are implicitly packed in Pascal/Z, therefore PACK and UNPACK are not permitted.

INPUT/OUTPUT: 'INPUT' and 'OUTPUT' are not allowed in the program heading, as input and output are implicit in Pascal/Z. However, INPUT and OUTPUT are not assumed as the default files, as they are in other implementations. Pascal/Z uses its modified I/O routines to cover all ranges of input and output (see PASCAL/Z INPUT AND OUTPUT).

POINTERS: Pascal/Z implements pointers using NEW, MARK and RELEASE, rather than the NEW

and DISPOSE described by Jensen & Wirth. This is done to maintain compatibility with most existing Pascal compilers. (See PASCAL/Z POINTERS for more detail.)

READ/WRITE: Each parameter to a READ/WRITE call of a text file should have a size less than 256 bytes.

LARGEST PROGRAM: The largest program which can run under Pascal/Z is limited by memory size and by the linker, since LINK/Z must be resident in memory at link time.

LIMITATIONS

The limitations of Pascal/Z are as follows:

* No passing of procedures or functions as parameters.

* Variable declarations limited to 32K per group (i.e., no procedure/function or main program can declare more than 32K of local storage.

* In order for the types of two records or arrays to match, they must be the same size (see the section on MEMORY USAGE). The following example should help to clarify this:

These do not match:
```
     X : ARRAY[ 1..10 ] of 0..255;
     Y : ARRAY[ 1..10 ] of INTEGER;
```
because X requires 10 bytes of storage and Y requires 20.

These do match:
```
     X : ARRAY[ 1..10 ] of INTEGER;
     Y : ARRAY[ 1..10 ] of 0..1000;
```
because both X and Y require 20 bytes of storage.

These do match:
```
     X : 0..255;
     Y : INTEGER;
```
because X and Y are not records or arrays.

The compiler flags as errors operations on types which do not match.

See the section on PASCAL/Z TYPE DECLARATIONS for more information.

* Standard GET/PUT input/output is not implemented. Pascal/Z uses its modified READ/WRITE facilities to handle all input/output. See PASCAL/Z INPUT AND OUTPUT for more detail.

* The PAGE routine is not implemented. (This routine is being considered for elimination from the proposed ISO standard.)

COMPILER OPTIONS

There are twelve switchable compiler options which may be enabled/disabled at any point in the compilation. Compiler options are specified in the first part of any comment. The format for enabling/disabling compiler options is:

(*$x+,y-,... <any comment> *)

where x, y, ... are any of the compiler options described below and the '+' means enable (turn ON or leave ON) and the '-' means disable (turn OFF or leave OFF).

NOTE: Do not use more than one dollar sign ($) in the same comment -- it will cause a disaster. Also, there should be no space or other character between the opening bracket ({) and the dollar sign--in such an instance the compiler will ignore the intended compiler options and the entire comment.

The default for each of the options is ON unless otherwise specified. The options are:

C Indicates that the compiler should generate code to check for CTRL-C (typed at the keyboard) before every GOTO statement and at the bottom of every REPEAT, WHILE and FOR loop. If this option is enabled and a CTRL-C is detected all open output files will be closed and program execution will be terminated.

E Indicates that the compiler should generate code to include the statement number of the statement being executed if there is a run-time error. Default for extended run-time error messages is OFF.

F Indicates that the compiler should generate code to check for floating point (REAL) over/underflow errors.

I Imbed Pascal source statements in the macrocode output of the compiler. Default for this option is OFF.
(NOTE: $I when used in a comment can also mean INCLUDE. See the section on INCLUDE FILES and be aware of the difference.)

J This option is used to generate a particular type of CASE statement. For information about its use see the section PASCAL/Z CASE STATEMENT. NOTE: This must be the last option specified in an options list.

L Indicates that the Pascal source program should be listed. Lines with errors are always listed. Each time that this option is used a new listing page is started; so this option can be used to paginate the listing of your Pascal programs by repeatedly turning this option ON.

M Indicates that the compiler should generate the code necessary to check for INTEGER multiply and divide errors. If an error is detected a message is printed and the program execution is terminated. Addition and subtraction errors are ignored.

P Allows symbolic Input/Output of enumeration types declared while this option is enabled.

R Indicates that the compiler should generate code to do range and bounds checking. If enabled this option generates code to detect range (assignment out of range) and bounds (array index out of range) errors.

S Enables stack overflow checking during procedure/function entry.

T Trace program execution. Each time a statement compiled with this option is executed the statement number is printed on the console device. Default setting for tracing is OFF.

U Indicates that the compiler should generate code to do range and bounds checking of parameters passed to user routines. Default is ON. This option will work only if the R option is enabled. WARNING: This option will check only value parameters.

Below is our sample program with some compiler options added:

```
SAMPLE                                                    Page 1


   1    1    0     PROGRAM Sample; (*$e+ extended error messages *)
   3    1    1       VAR  I, J : COLOR;
   4    1    1            S1, S2 : ARRAY[ 1..5 ] OF CHAR;
   5    1    1     BEGIN {$i+ imbed the Pascal source in the
                             macro-code }
   6    1    1           FOR I := RED TO YELLOW DO
   7    2    2             WRITELN( 'COLOR IS:', ' ', I : 1 );
   8    3    1           {$1+,i- new listing page, stop imbedding }

SAMPLE                                                    Page 2


   9    3    1           S1 := 'FIRST';
  10    4    1           S2 := 'LAST ';
  11    5    1           WRITE( S1, S2 );
  12    6    1     END.
```

OPTIMIZING PASCAL/Z PROGRAMS FOR SPEED

This is a guideline for writing faster programs in Pascal/Z.

General Research by Niklaus Wirth has shown that greater than 90% of all variable accesses in most programs are to either local or global data areas. In Pascal/Z variable access has been optimized for accesses to the local and global levels, therefore greatly increasing the efficiency of variable accesses (by approximately four times).

Constants With the exception of SETs, it is always faster to use constants than it is to use variables. Declared string constants are more efficient (by one relative jump) than in-line string constants.

INTEGERs It is usually faster to use one byte INTEGERs instead of two byte INTEGERs when possible. However, if range checking is enabled, assignments to subranges take longer than assignments to INTEGERs.

RECORDs In Pascal/Z the use of the WITH statement does not affect execution speed unless the RECORD is "dynamic". A RECORD is "dynamic" if:

1) it is a member of an ARRAY with something other than a constant for the index (i.e., A[I] as opposed to A[3])

2) it is a parameter passed by reference.

If the RECORD is "dynamic" then the use of the WITH statement increases the program speed in case 1 and decreases the speed in case 2.

ARRAYs When accessing ARRAYs it is possible to get at an element more quickly if the index (or any of the indices, if there are more than one) is a constant. Also range checking (which is only done at compile-time for constant indices), can slow down ARRAY accesses considerably.

CASE It is always faster to use a CASE statement instead of a list of two or more IF statements.

Number Crunching When using constants it is MUCH MUCH MUCH faster to use REAL constants in assignments to REALs rather than to use INTEGER constants which must be implicitly converted to REAL (e.g., n:=n*6.0 is much faster than n:=n*6 if n is a REAL number).

PASCAL/Z TYPE DECLARATIONS

When declaring types in Pascal/Z, there are certain methods of declaration which result not only in more readable code, but also in less use of the type table. This section contains a few hints on how to declare types so as to utilize a minimum of type table space. Efficient type declarations also result in faster compilation times.

In Pascal, any two or more types may be compared for STRUCTURED EQUIVALENCE or for NAMED EQUIVALENCE. Structured equivalents are the same in structure, but are declared in different type declarations. Named equivalents are the same in structure and are also named within the same type declaration. Thus if two types are named equivalents, they must also be structured equivalents, but not necessarily vice versa.

The following examples should help to illustrate the idea of structured and named equivalents.

EXAMPLE #1:

```
type employee_name = array[ 1..30 ] of char;

var  x  :  employee_name;
       y : real;
       z : array[ 1..30 ] of char;
```

In the above declarations, X and Z are STRUCTURED equivalents, but are not named equivalents, because although X and Z are structurally the same, two separate type declarations are made.

In Example #1,

```
type employee_name = array[ 1..30 ] of char;
```

is an explicit type declaration, and it results in one entry in the type table. The size of the entry will be determined by the complexity of the type declared.

Also in Example #1,

```
var  z  :  array[ 1..30 ]  of  char;
```

is an implicit type declaration, and it results in an additional entry in the type table, which could have been avoided by declaring it as follows:

```
var   z   :   employee_name;
```

Since employee_name was already declared as a type of array[1..30] of char, using the expression again is redundant.

EXAMPLE #2:

```
type employeeinfo = record
                      name : array[ 1..30 ] of char;
                      age : integer;
                      sex : ( m, f );
                      salary : real
                    end;          { employeeinfo }

var x : employeeinfo;
    y : employeeinfo;
```

The above variables are STRUCTURED equivalents, because X and Y are identical in structure. Since X and Y are declared within the same type declaration, they are also NAMED equivalents.

Pascal/Z allows both STRUCTURED and NAMED equivalents. However, in Pascal/Z it is always more efficient to use named equivalents.

HOW TO RUN PASCAL/Z

The following files are necessary to compile, assemble, and link an ordinary Pascal/Z program (one which makes no use of special features such as external routines or separate compilation):

```
<your program>
PASCAL48.COM
PAS248
DECS
PFSTAT
(or PASCAL54.COM and PAS254)
ASMBL.COM
MAIN.SRC
LINK.COM
LIB.REL
```

To run the Pascal/Z compiler, make sure that the necessary files are on the currently logged-in drive and type:

PASCAL48 <filename>.<source drive><output drive><listing drive>

where,

<filename>	is the file name of the text file with the extension .PAS
<source drive>	is the letter naming the drive that the source file is on. The currently logged-in drive is the default.
<output drive>	is the letter naming the drive to which the Z-80 macro-code generated by the compiler should be sent. The currently logged-in drive is the default.
<listing drive>	is the letter naming the drive to which the Pascal listing should go. The currently logged-in drive is the default.

A space in place of an option letter specifies the default.

EXAMPLES

```
A>PASCAL48 PRIMES
```

This will compile the text file PRIMES.PAS on drive A. PRIMES.LST (the program listing) and PRIMES.SRC (the Z-80 macro-code) will be sent to drive A.

```
B>PASCAL48 PRIMES.ABC
```

This will compile the text file PRIMES.PAS on drive A. PRIMES.SRC will be sent to drive B, and PRIMES.LST will be sent to drive C.

```
B>PASCAL48 PRIMES.bAb { where a "b" represents a blank }
```

This will compile the text file PRIMES.PAS on drive B (the default). PRIMES.SRC will be sent to drive A. The listing file will be sent to drive B (the default).

The listing (.LST file) of a Pascal/Z program may also be sent directly to the console or the printer during compilation by replacing the third letter of the file extension with "x" to specify the console and "y" the printer.

EXAMPLE

```
A>PASCAL48 PRIMES.AAX
```

This will compile the text file PRIMES.PAS on drive A. PRIMES.SRC (the Z-80 macro-code) will be sent to drive A, and the listing file will be sent to the console.

```
A>PASCAL48 PRIMES.BBY
```

This will compile the text file PRIMES.PAS on drive B. PRIMES.SRC will be sent to drive B, and the listing file will be sent to the printer.

While compiling, the console output of Pascal/Z is as follows. Every time a procedure/function declaration is encountered, its name and the present line number are printed. A '-' is output every time that a new procedure or function declaration is encountered and every time that 10 lines have been compiled. An 'E' is output instead of a '-' if there was an error in the last group of lines (not necessarily 10 lines depending on procedure/function declarations). When compilation is completed, the total number of errors is output to the console. (NOTE: The errors are listed individually in the .LST file, but the total number of errors is not.) The compiler may be stopped at any time by typing CTRL-C.

After compilation, a Pascal/Z program must be assembled and linked before it may be run.

Before assembly, however, you may wish to optimize the output of the compiler by running it through PASOPT, a peep-hole optimizer designed for use only with the code generated by the Pascal/Z compiler. To invoke the optimizer, type:

```
A>PASOPT PRIMES.SRC
```

The optimizer will scan through the Z-80 source file and reorder certain patterns in the code to make them more efficient. The optimizer will rename the old source file to PRIMES.ORG (for original) and generate a new optimized file called PRIMES.SRC. When it has finished optimizing, a message giving the total number of bytes eliminated will appear.

Note that since PASOPT has been developed specifically for use with the output of Pascal/Z, running any ordinary assembly language programs through it will cause false (and usually fatal)

results. Therefore, do not run any file through the optimizer more than once.

To assemble a Pascal/Z program, type:

```
A>ASMBL MAIN,PRIMES/REL
```

The console will display:

```
Pascal/Z run-time support interface ASMBLE v-7c
```

This command will cause the assembler (ASMBLE/Z) to automatically search for the .SRC file associated with the filenames MAIN and PRIMES. MAIN.SRC is a file containing definitions and routines to be assembled with the output of the compiler, and MUST ALWAYS BE THE FIRST FILENAME SPECIFIED.

ASMBLE/Z will assemble the Z-80 macro-code and output a relocatable object code module, as specified by the extension .REL. When assembly is successfully completed, the following message appear:

```
0 errors. X symbols generated. Space for X more symbols.
X characters stored in X macros.
X bytes of program code.
```

where X is an integer number.

The next step is to link the program with any necessary subroutines.

(If you wish to use the Pascal/Z interactive symbolic debugger, InterPEST [InterSystems Pascal Error Solving Tool], see the InterPEST Reference Manual for details on linking Pascal/Z programs.)

To do so, type:

```
A>LINK PRIMES/N:PRIMES/E
```

The console will display:

```
LINK version 2b
Load mode
Generate a COM file
```

This command will cause the linker (LINK/Z) to automatically link the file PRIMES.REL with the library subroutines contained in LIB.REL, provided on the distribution diskette. Only the library modules which are called by the Pascal/Z program PRIMES will be linked in, reducing the size of the final code.

The /N:PRIMES option specifies that the linker should generate a .COM file with the name PRIMES.

The /E option indicates that after generating a .COM file, the linker should exit and return to CP/M.

When the link is completed, the console will display the following:

```
Lo = X Hi = X Start = X Save X blocks
```

The first three Xs will be hex addresses, and the last X will be an integer.

To run the program, simply type:

```
A>PRIMES
```

The command

```
A>LINK PRIMES/N:PRIMES/G
```

will yield the same results, except that instead of returning to the operating system, the program will be executed as soon as the link is finished, as specified by /G, or "go".

(For more detailed information on the assembler and the linker, see the accompanying manuals ASMBLE/Z and LINK/Z.)

* COMPILE.SUB

A Pascal/Z program may be compiled, assembled, linked and run automatically by using the COMPILE.SUB file on the distribution diskette. PASCAL48.COM, PAS248, PFSTAT, DECS, MAIN.SRC, LIB.REL, ASMBL.COM, LINK.COM, COMPILE.SUB and SUBMIT.COM must be on the diskette in drive A. Note that although the version of COMPILE.SUB supplied on the diskette uses the 48K overlaying version of the compiler, those users who wish to use the 54K version may change COMPILE.SUB quite simply by editing the file and changing PASCAL48 to PASCAL54 (BZ users may wish to change it to PASCALBZ). NOTE: If you change the invocation command, be certain that the necessary files are on the logged-in drive (i.e. PASCAL54.COM needs PAS254, etc.).

The proper command to submit the Pascal/Z program for processing is:

```
SUBMIT COMPILE <your program name> X
```

followed by a carriage return, where X is the letter of the drive containing the diskette which your program is on. All files (.SRC, .LST, .REL, .COM) will be sent to the drive specified by X. If the drive letter is not specified, the process will be halted at link time, and the message "Can't find <program>.REL" will be displayed.

Note that once the COMPILE.SUB file is submitted, processing will continue through the execution of your program regardless of any errors which may be present. The submit file can be halted by typing CTRL C twice. It will finish executing the current command of the COMPILE.SUB file and then return to the operating system.

INTERPRETING PASCAL/Z LISTINGS AND ERROR MESSAGES

When your Pascal program is compiled, the compiler generates a listing file of your program which includes pagination, line, statement, and nesting level numbers, as well as any compilation errors (.LST file). The number to the far left of each line is the line number. The next number is the number of the first statement on that line (if there is no statement on that line then the same number will appear on the next line). Statement numbers are used in conjunction with the trace and extended error message options (see section on PASCAL/Z COMPILER OPTIONS). The other number preceding each line is the number of levels that that statement is nested.

Here is a sample listing of a program with no errors:

```
SAMPLE                                                    Page 1


Line Stmt Level
   1    1    0     PROGRAM Sample;
   2    1    0       TYPE COLOR = ( RED, BLUE, GREEN, YELLOW );
   3    1    1       VAR  I, J : COLOR;
   4    1    1            S1, S2 : ARRAY[ 1..5 ] OF CHAR;
   5    1    1     BEGIN
   6    1    1            FOR I := RED TO YELLOW DO
   7    2    2              WRITELN( 'COLOR IS:', ' ', I : 1 );
   8    3    1            S1 := 'FIRST';
   9    4    1            S2 := 'LAST ';
  10    5    1            WRITE( S1, S2 );
  11    6    1     END.
```

In the unlikely event that you have written a program which generates compilation errors, the program listing will show the type and location of the error(s). If the error is a syntax error, the error message will be given in English and a ^ will mark the location of the error. If the error is a semantic error then the error message will be one of the compile-time error codes listed on pages 119-121 of the PASCAL USER MANUAL AND REPORT, as well as in Appendix Six at the back of this manual. In addition to the standard error codes listed in the manual, Pascal/Z has further defined the implementation restriction error (398) as follows:

3980 Symbol table overflow (one way to help avoid this error is to minimize the number of FORWARD declared procedures and functions).

-3980 Type table overflow (One way to avoid this is to minimize the number of type declarations. See the section on PASCAL/Z TYPE DECLARATIONS.).

3981 Function value may not be qualified.

3982 Jump out of a procedure/function not allowed in Pascal/Z.

3983 Non-string compared with string.

3984 Program has too many levels of nesting. Depends upon the complexity of the user's program.

-3984 No more than forty fields in a record. (Can be avoided by nesting records.)

3985 Can't output/input this value because compiler option P was disabled when this enumeration type was declared.

3986 Line or symbol too long. The maximum line/symbol length is eighty characters.

3987 Maximum string length is 255.

3988 String too small for call by reference.

-3988 Declarations of BCD numbers passed by reference must match exactly. (Only generated when using Pascal/BZ.)

3989 (This error message has changed from Version 3.2-1. Structured values returned by functions are now allowed as parameters to a WRITE or WRITELN.)

The error message 3989 now indicates that an EXTERNAL was declared in a separate module. All EXTERNALs must be declared in the main program.

One of the most common error messages is 'Program too complex'; this error message will appear on the console at compile-time, and compilation will be halted. This message usually means there is not enough memory in the system to run the compiler. If using the 54K version, try switching to the 48K version. Otherwise, adding memory to the system will usually solve the problem. It may also mean that there is not enough stack space remaining. To remedy this, stack usage must be reduced before trying to recompile.

The message "Too many errors" means that an integer constant is out of range of the allowable integer values.

Another error encountered when using separate compilation is 'premature EOF'. This generally means that the CP/M file names of the modules do not correspond to the internal module names as specified in the module headings. The problem is easily solved by checking to make certain the names correspond. This error will also occur if the fourth drive letter is not specified when compiling the program modules.

* RUN-TIME ERRORS -- STACK OVERFLOW

One of the most common run-time errors is stack overflow. This indicates that the stack space has been exhausted, and is a fatal error. One way of avoiding this is to keep the number of nesting levels to a minimum. Also, if the same parameters are passed to a number of different routines, it may be possible to declare them globally, and thus reduce stack usage.

Here is the same program as above with a few errors added:

```
SAMPLE                                                    Page 1


Line Stmt Level
   1    1     0     PROGRAM Sample;
   2    1     0       TYPE COLOR = ( RED, BLUE, GREEN, YELLOW );
   3    1     1       VAR  I, J : COLOR;
   4    1     1            S1, S2 : ARRAY[ 1..5  OF CHAR;
                                       ^ ]  EXPECTED
   5    1     1     BEGIN
   6    1     1            FOR I := RED  YELLOW DO
                                 ^ TO          EXPECTED
   7    2     2              WRITELN( 'COLOR IS:', ' ', I : 1 );
   8    3     1            S1 := 'FIRST';
   9    4     1            S2 = 'LAST ';
                           ^ := EXPECTED
  10    5     1            WRITE( S1, S3 );
                                   ^ ERROR 104
                                     ^ ERROR 116
  11    6     1     END.
```

Since Pascal/Z is a recursive descent compiler, errors are sometimes not discovered until a couple of symbols after the symbol which is erroneous; this is especially likely to occur with multi-line statements. The ^ mark in the error descriptions is placed as close as possible to the symbol which is in error (for accurate placement of ^ refrain from using tabs). Also sometimes errors "percolate" through a program (i.e., one error may cause the compiler to become "confused" and generate additional errors even if there is nothing wrong); in these cases the first error is the one which should be believed.

PASCAL/Z INPUT AND OUTPUT

All input/output and related operations in Pascal/Z are done with eight standard routines: RESET, REWRITE, READ, READLN, WRITE, WRITELN, EOLN, and EOF. These routines allow the user to create, delete, read, write, and test the status of operating system files. Although these routines differ slightly from the standard Pascal I/O routines, they are more flexible and are easier to use. These routines are the only way to access file data; GET and PUT (and also <fvar>^) may not be used. In this table <fnam> is any legal Pascal file name, <fvar> is any legal Pascal file variable and anything enclosed in square brackets is optional.

RESET(<fnam>, <fvar>)

RESET is used to reset an input file to the beginning of the file and open it for access. <fnam> is any legal operating system filename (may be a quoted string , an ARRAY OF CHAR or a STRING), and <fvar> is the Pascal/Z file variable. A RESET must be done before any non-console input file may be read.

REWRITE(<fnam>, <fvar>)

REWRITE is used to open a file for output. If the file already exists then the old file is deleted before the new file is opened. A REWRITE must be done before any non-console file is written to. (See section on DEVICE INPUT AND OUTPUT for device output specification.)

Note that the return value of a function may not be used to specify a file name in a RESET or REWRITE statement [e.g. RESET(<function return value>, fvar) is not allowed].

READ([<fvar>,] p1, ..., pn)

READ reads the previously opened input file and stores the information in the variables specified. The file may or may not be of type TEXT. For TEXT files the allowable parameter types are INTEGER, CHAR, BOOLEAN, ARRAY OF CHAR, REAL and correctly declared enumeration types. The number of parameters is variable, but at least one must be used. If the <fvar> parameter and trailing comma are omitted, the console is used for input (the console file is of type TEXT).

READLN([<fvar>,] p1, ..., pn)

READLN is the same as READ except that a new line is found after the READ is finished. READLN should only be used on TEXT files.

WRITE([<fvar>,] p1, ..., pn)

WRITE is used to write into a previously opened output file. If the file is of type TEXT, then the parameters may be INTEGER, BOOLEAN, CHAR,

ARRAY OF CHAR (including quoted strings), REAL, and any properly declared enumeration type. As in READ and READLN, the number of parameters is variable, but must be greater than zero. If the <fvar> parameter is ommitted, then the console is used for output (type TEXT).

WRITELN([<fvar>,] p1, ..., pn)
WRITELN is the same as WRITE except that it should only be used to write into files of type TEXT, and it appends a carriage return/line feed to the data being output. If WRITELN is used with no parameters, a blank line is written to the console.

EOLN(<fvar>)
EOLN is a BOOLEAN function defined for input files of type TEXT and returns TRUE if the file indicated is at the end of a line. EOLN(0) returns TRUE if there is no more input in the console input buffer.

EOF(<fvar>)
EOF is a BOOLEAN function which returns TRUE if the specified input file is exhausted or if there has been any type of operating system file error (i.e., file not found, read error, etc.) A note of caution --- Because CP/M does not keep any information regarding partially filled blocks at the end of a non-text file, it is impossible to make EOF(<non-text file>) work correctly unless the record size used is a multiple of 128. The suggested way of working around this problem is to either know how many records are in the file or to have a special end-of-file record configuration.

Each parameter to a READ/WRITE call of a text file should have a size less than 256 bytes.

As the compiler is shipped to you, the maximum permissible number of simultaneously open output files is four. However, the user may increase this number be redefining MAXOUT in MAIN.SRC (the time required to exit a procedure/function increases as MAXOUT increases).

There is no way of explicitly closing a file; however, whenever the block containing a file buffer is exitted (or if the program terminates) the file will be closed. If a file buffer is used in another RESET, the file being RESET will be closed prior to the opening of the second file.

After starting a Pascal/Z program the CP/M command tail is stored in the input buffer and may be read by subsequent read requests. Pascal/Z uses the console input buffer to build file names when opening a disk file. Therefore, if there is some information in the console input buffer which you would like to read, read it before opening any files.

The following example programs demonstrate the different uses of the Pascal/Z I/O routines. The first example reads input from the console and writes it back to the console. The second uses file I/O, and takes input from the file named 'data', then writes the output to the file named 'result'.

```
Program IO;

{This is a simple program to demonstrate Pascal/Z Input/Output.}
{The following is an example of how the READ, READLN and WRITE,}
{WRITELN statements work.                                      }
{READ will read input, in this case from the console.          }
{READLN will do an initial read, and then skip to the next line}
{of input.                                                     }
{WRITE will write output, in this case to the console.         }
{WRITELN will do an initial write, and then append a carriage  }
{return, line feed to the specified output.                    }

var     x, y : integer;         { input data     }
        i : integer;            { index variable }

begin     { the following block demonstrates the use of READ, }
     { WRITE and WRITELN.                                      }

     for i := 1 to 3 do
          begin
               read( x, y );
               writeln( 'The numbers are:', x : 5, y : 5 );
               write( 'The sum is: ', x + y );
               writeln( '   The difference is: ', x - y );
               writeln
          end;

     { the following block demonstrates the use of READLN, }
     { WRITE and WRITELN.                                   }

     for i := 1 to 3 do
          begin
               readln( x, y );
               writeln( 'The numbers are:', x : 5, y : 5 );
               write( 'The sum is: ', x + y );
               writeln( '   The difference is: ', x - y );
               writeln
          end
end.
```

When the program is run and the following data is input from the console:

```
354 26 492 1032 783 97 564 928 37
471 216 34 841
1985 672 47
```

the following results will be obtained:

```
354 26 492 1032 783 97 564 928 37
The numbers are:  354   26
```

```
The sum is:         380   The difference is:         328

The numbers are:  492 1032
The sum is:        1524   The difference is:        -540

The numbers are:  783   97
The sum is:        880   The difference is:         686

The numbers are:  564  928
The sum is:       1492   The difference is:        -364

471 216 34 841
The numbers are:  471  216
The sum is:        687   The difference is:         255

1985 672 47
The numbers are: 1985  672
The sum is:       2657   The difference is:        1313
```

```
Program Fileio;

{ This is a program to demonstrate file I/O using Pascal/Z. }
{ Pascal/Z does not allow the use of GET and PUT, but does  }
{ all input/output utilizing its modified READ / WRITE and  }
{ RESET and REWRITE routines.                               }

var     x, y : integer;          { input data                }
     infile : text;           { data input file variable  }
     outfile : text;      { data output file variable }

{ In this program, data is read from an input file 'data' and }
{ the results of the program are written to an output file    }
{ 'result'.                                                   }

begin
     { reset the input file 'data' to the beginning }
     reset( 'data', infile );
     { open the file 'result' for output from the program }
     rewrite( 'result', outfile );
     while not eof( infile ) do
          begin
               readln( infile, x, y );
               writeln( outfile, 'The numbers are:', x : 5, y : 5 );
               write( outfile, 'The sum is: ', x + y );
               writeln( outfile, '   The difference is: ', x - y );
               writeln( outfile )
          end
end.
```

With the following data contained in the input file 'data':

```
354 26 492 1032 783 97 564 928 37
471 216 34 841
1985 672 47
```

the results will be as follows, and will be contained in the output file 'result':

```
The numbers are:  354   26
The sum is:        380   The difference is:        328

The numbers are:  471  216
The sum is:        687   The difference is:        255

The numbers are: 1985  672
The sum is:       2657   The difference is:       1313
```

A description of how to use PASCAL/Z INPUT AND OUTPUT with Direct Access Files is included in the section on DIRECT FILE ACCESS.

DIRECT FILE ACCESS

Pascal/Z provides the user with a Direct Access Facility which is used to directly read or write any record in a file. The term "direct access" is used rather than random access since all file accesses are not equal. Moving a disk head from track 1 to track 2 is much faster than moving it to track 50; such accesses are not truly random.

The syntax for these direct reads and writes is as follows:

DIRECT WRITE:
    WRITE(<fvar>:<record number>, <record variable>)

DIRECT READ:
    READ(<fvar>:<record number>, <record variable>)

where the record variable is a Pascal record variable.

EXAMPLES:

Write a record to position 35 in file Q:
     WRITE(Q:35, <record variable>)

Read record I from file Q:
     READ(Q:I, <record variable>)

The records and the record numbers correspond directly; READ(Q:1, <record variable>) reads the first record in the file, READ(Q:12, <record variable>) reads the twelfth record in the file.

If after a file has been repositioned a subsequent read or write does not specify a record position, then the next sequential record is assumed.

All files start at logical record one; an attempted access of logical record zero will access the next sequential record. If a write is attempted beyond the current EOF then the file is extended to that point before the write is performed.

REWRITE is used to create a new file. If a file with the same attributes already exists it is deleted before the new file is created.

RESET is used to work with an existing file. A file which has been reset may still be written to using random writes.

Once a file has been randomly accessed, all subsequent sequential reads and writes will be much slower (due to CP/M's aversion to mixing random and sequential access) until the file has been closed and re-opened.

Random access is implemented in, and thus only supported with, CP/M 2.0 or higher.

RENAME AND ERASE

RENAME and ERASE are two external Pascal routines which allow a Pascal program to RENAME and ERASE files from the file directory. In effect they provide a clean interface to the CP/M system functions RENAME (23) and DELETE (19).

To incorporate these functions in a Pascal program the following declarations are required:

```
Type   filestring   =   string   14;

Function RENAME (oldfile, newfile: filestring):
                              Boolean ; External;

Function ERASE (oldfile: filestring): Boolean; External;
```

Oldfile contains an unambiguous file name (ufn) of an existing CP/M file. Newfile also contains a ufn. NOTE that when using the Pascal external RENAME function, the old file name comes first, and the new file name second, unlike the CP/M RENAME facility.

Each function returns TRUE if its operation was successful, FALSE otherwise. Failures can result from files not being found or from illegal file names.

The user should also be sure that all files affected are closed when a RENAME or ERASE is attempted.

Examples:

```
if  RENAME ('P.COM', 'PIP.COM')
     then writeln ('PIP');

if ERASE  ( 'MASTER.BAK' )
     then writeln( 'master.bak has been deleted' );
```

RENAME and ERASE are in the LIB.REL file and are automatically linked by LINK/Z at link-time. The Z-80 source code for the RENAME and ERASE routines are contained in the file RENERA.SRC. It is found on the same side of the disk containing the fixed point package.

Two sample programs RENDRV.PAS and DELDRV.PAS are provided which drive RENAME and ERASE. After compiling and linking they may be run to rename or erase files from the console (erase with caution!). Exit either program by typing ^C instead of a file name.

DEVICE INPUT AND OUTPUT

INPUT

Pascal/Z allows the user to specify input from the console using CON:.

To input from the console:

```
RESET( 'CON:', <fvar> );
READ( <fvar>, Pascal variable );
```

OUTPUT

Pascal/Z allows the user to specify output to the console or to a printer using CON: or LST:. To access the console or printer use:

```
REWRITE(  'CON:',  <fvar>  );  {  console  }
REWRITE(  'LST:',  <fvar>  );  {  printer  }
```

and then write to the device the same way you would to any other output file. For example, the following program copies a text file to the console.

```
PROGRAM CONSOLE;
  TYPE STR = STRING 80;
  VAR FNAM, FOUT : TEXT;
      LINE : STR;
  BEGIN
    RESET( 'TEST.TXT', FNAM );
    REWRITE( 'CON:', FOUT );
    WHILE NOT EOF( FNAM ) DO BEGIN
      READLN( FNAM, LINE );
      WRITELN( FOUT, LINE )
      END
  END.
```

When using Pascal/Z device input and output, the file variable (fvar) must be declared as type TEXT.

PASCAL/Z EXTENSIONS

While we at InterSystems feel strongly that it is important to keep Pascal extensions to a minimum in order to maintain simplicity, clarity and transportability, we also feel that in order for Pascal to be useful in the business and scientific communities a few necessary extensions must be made. The extensions presented in Pascal/Z represent an attempt to extend the utility of Pascal while maintaining the "spirit" of the language.

Each of the extensions in Pascal/Z is listed here along with a justification for including it in the language. Full descriptions of how to use the extensions, along with examples, are given in the appropriate sections later in this manual.

1) Pascal/Z allows certain types of expressions where Pascal requires a constant to be used. This allows the user to declare a series of inter-related constants where it is only necessary to change one constant when a change is desired.

2) Another extension made to Pascal/Z is to allow functions to return structured, as well as scalar, values. This greatly increases the range of applications for functions.

3) A variable length string type has been added to Pascal/Z. It is extremely useful for text manipulation and for certain types of I/O.

4) There is now an ELSE clause for the CASE statement. This is one of the most common extensions to Pascal (sometimes called OTHERS or OTHERWISE) and is extremely useful when it is desirable to use the power of the CASE statement without listing all of the possible cases.

5) Separate compilation is permitted. This decreases program development time by allowing the user to divide his program into distinct modules, which can then be debugged, compiled and assembled separately, avoiding the need to recompile and reassemble the entire program.

6) Pascal/Z provides the user with a facility to link EXTERNAL routines to Pascal programs. These routines allow the user to communicate directly with I/O devices as well as to perform operations which are more naturally done in assembly language.

7) Overlay capabilities are supported to permit development of programs larger than system memory size. This facility is extremely useful for the software developer who may be forced to limit program function to accommodate system restrictions.

8) Pascal/Z supports INCLUDE files to allow the user to insert a file at any point in a Pascal program. This is useful in decreasing typing and editing time, since

program blocks which are used frequently can be contained in a unique file, and then INCLUDED in the Pascal/Z program.

9) One of the contradictions in Pascal is the ability to symbolically input/output the type BOOLEAN, which is defined as an enumeration type

```
BOOLEAN = ( FALSE, TRUE )
```

but not to be allowed to symbolically input/output the values of any other (i.e., user declared) enumeration types. In Pascal/Z the user is allowed to symbolically input/output the values of any enumeration type.

10) It is often useful to be able to directly access any record in a FILE. Pascal/Z has added Direct File Access (sometimes called random access in other implementations) to allow this to be done in a reasonable fashion.

PASCAL/Z CONSTANTS

The first Pascal/Z extension is the redefinition of a constant. In the USER MANUAL AND REPORT a constant is defined as:

```
<constant> ::= <unsigned number> | <sign> <unsigned number> |
               <constant identifier> |
               <sign> <constant identifier> | <string>
```

In Pascal/Z a constant is defined as:

```
<new constant> ::= <constant> |
                   <integer constant> * <integer constant> |
                   <integer constant> + <integer constant> |
                   <integer constant> - <integer constant> |
                   <integer constant> div <integer constant>
```

This is extremely useful for the following type of declaration:

```
CONST CLASS = 30;            { number of students }
      GIRLS = 17;            { number of girls }
      BOYS = CLASS - GIRLS; { number of boys }
```

PASCAL/Z FUNCTIONS

Pascal/Z versions 3.2 and later allow functions to return structured values, as well as scalar values. Functions returning structured values are declared in the same manner as functions returning scalar values, the only difference being in the specification of the return value.

For example;

```
TYPE NUMBERS = ARRAY[1..100] OF INTEGER;
       NAME = STRING 20;
VAR  NUM : NUMBERS;
      ID : NAME;

{A FUNCTION TO INCREMENT EACH ELEMENT OF AN ARRAY}
FUNCTION INCREMENT( CNT:NUMBERS ): NUMBERS;
  VAR X : INTEGER;
  BEGIN
    FOR X := 1 TO 100 DO CNT[X] := CNT[X] + 1;
    INCREMENT := CNT
  END;

{A FUNCTION TO CREATE AN ARRAY WITH EACH}
 {ELEMENT 20 GREATER THAN THE OLD ARRAY}
FUNCTION XXPLUS( CNT:NUMBERS ): NUMBERS;
  VAR X : INTEGER;
  BEGIN
    FOR X := 1 TO 100 DO
    XXPLUS[X] := CNT[X] + 20
  END;

{A FUNCTION TO APPEND A ' ESQ.' TO A NAME}
FUNCTION ADDESQ( VAR PERSON:NAME ):NAME;
  BEGIN
    IF LENGTH( PERSON ) <= 15 THEN
      APPEND( PERSON, ' ESQ.' );
    ADDESQ := PERSON
  END;

BEGIN
  .....
  NUM := INCREMENT( NUM );
  .....
  NUM := XXPLUS( NUM );
  .....
  ID := ADDESQ( ID );
```

NOTE: In the third example, the function LENGTH must be declared as shown in the section on STRINGS.

In the example above of function INCREMENT, the entire array CNT was assigned to INCREMENT; while in XXPLUS, each element was individually assigned. Both of these formats are legal within the function. However, an element of a function returning an array (or a field of a function returning a record), cannot appear on the right side of an assignment statement. The entire

value of the return value must be assigned to a variable of the defined return type.

```
NUM := INCREMENT( NUM ); {LEGAL STATEMENT}

NUM[ 10 ] := INCREMENT( NUM[ 10 ] ); {ILLEGAL}
```

The entire return value of a function can also be passed by value as a parameter, or compared using relational operators.

PASCAL/Z STRINGS

In addition to the standard Pascal string (see Pascal User Manual and Report, pages 40-41), Pascal/Z introduces variable length strings to the Pascal language. While this extension is the least Pascal-like of all Pascal/Z extensions, it is none the less quite useful for programmers who are used to the availability of such types.

Variable length strings are declared using the reserved word STRING followed by the maximum length of that string (maximum allowable string length is 255).

```
VAR Z: STRING 126; { string with a maximum length of 126 }
```

There is also an additional intrinsic procedure APPEND. APPEND is used to append one string to another; for example:

```
APPEND( Z, '!' ); { add '!' to Z }
APPEND( Z, ' FINI' ); { add ' FINI' to Z }
APPEND( Z, Z );     { add Z to Z }
```

APPEND was implemented rather than a concatenate procedure because many string operations are appends and appending is more efficient than concatenating; however concatenate may be simulated as follows:

```
VAR TEMP: STRING 255;
BEGIN
    ....
    TEMP := Z;
    Z := '1) ';
    APPEND( Z, TEMP );  { append Z to 1) and store in Z }
```

There are other string routines in the Pascal Run-Time Support Library, but the programmer must declare them externally in order to access them. Before declaring these external routines the following types must be declared:

```
$STRING0 = STRING 0;
$STRING255 = STRING 255;
```

The routines and their declarations are:

```
FUNCTION LENGTH( X: $STRING255 ): INTEGER; EXTERNAL;
```
returns the present length of a string.

```
FUNCTION INDEX( X, Y: $STRING255 ): INTEGER; EXTERNAL;
```
returns the place in string X where the substring Y begins. If Y is not in X, a zero is returned.

```
PROCEDURE SETLENGTH( VAR X: $STRING0; Y: INTEGER ); EXTERNAL;
```
set the length of X to Y.

Strings may be initialized to the NULL string with the SETLENGTH routine.

The following restrictions apply when using the type STRING in Pascal/Z programs:

1) When passed by reference, the maximum length of a STRING must be greater than or equal to that of the formal parameter, hence the use of $STRING0 above in procedure SETLENGTH.

2) When passed by value the actual length of a STRING must be less than or equal to that of the formal parameter, hence the use of $STRING255 above in functions LENGTH and INDEX.

3) When used with relational operators a constant STRING may not appear to the left of the relational operator (generates compilation error 3989).

```
IF STRING1 = STRING2 THEN <statement>  { legal }
IF STRING1 = 'HELLO' THEN <statement>  { legal }
IF 'HELLO' = STRING1 THEN <statement>  { illegal }
```

It is possible to access the Nth character in a string using the form

```
Z[ N ]
```

The value of N will be checked to make sure that it is between 1 and the maximum length of Z (unless range checking has been turned off).

Note that at compile time a quoted string will take twelve bytes in the type table before it is written out. This space will be reallocated as soon as the statement containing the string has been executed.

Here is a short program using Pascal/Z STRING functions.

```
PROGRAM LONGLINE; {$I+ }
CONST LINESIZE = 80;
TYPE  $STRING0 = STRING 0;
      $STRING255 = STRING 255;
VAR   LINE: STRING LINESIZE;
      WORD: STRING 80;
FUNCTION LENGTH( X: $STRING255 ): INTEGER; EXTERNAL;
FUNCTION INDEX( X, Y: $STRING255 ): INTEGER; EXTERNAL;
PROCEDURE SETLENGTH( VAR X: $STRING0; Y: INTEGER ); EXTERNAL;
BEGIN
    WRITELN( 'TYPE ONE WORD AT A TIME AND THIS PROGRAM WILL',
             ' ASSEMBLE THE WORDS INTO LINES OF ',
             LINESIZE:1, ' WORDS EACH' );
    WRITELN( 'TYPE !"#>$ TO STOP' );
    SETLENGTH( WORD, 0 ); { INITIALIZE WORD TO NOTHING }
    REPEAT
        SETLENGTH( LINE, 0 ); { INITIALIZE LINE TO NOTHING }
        WHILE (LENGTH( LINE ) + LENGTH( WORD ) < LINESIZE)
              AND (INDEX( WORD, '!"#$' ) = 0) DO BEGIN
            APPEND( LINE, WORD );
            IF LENGTH( LINE ) < LINESIZE THEN
                APPEND( LINE, ' ' ); { WORD SPACE WORD }
            WRITE( 'THE WORD IS: ' );
            READLN( WORD );
        END;
        WRITELN( 'THE LINE IS:' );  WRITE( LINE );
    UNTIL INDEX( WORD, '!"#$' ) <> 0;
END.
```

PASCAL/Z CASE STATEMENT

As mentioned earlier an ELSE clause has been added to the CASE statement. The following example should help to clarify its use:

```
CASE COLOR OF
    RED, YELLOW, BLUE: PRIMARY := TRUE;
    ELSE: PRIMARY := FALSE
    END;
```

The CASE statement in Pascal/Z is usually implemented as a series of successive value tests, and while more efficient and neater than using IF statements, the benefit is not as large as it could be. Another method of implementing the CASE statement is with a jump table (i.e. using a table lookup to select the appropriate case); this is MUCH faster but sometimes requires a large jump table to work. For example, it would take 128 entries of 2 bytes each (total of 256 bytes) just for the jump table to have a CASE statement that used a jump table for CASE CH OF ..... where CH is of type CHAR. However, for some CASEs the required jump table is small or the speed gain is worth the table space and for this reason there is a compiler option to allow the generation of a jump table instead of individual value tests. The maximum table size allowed is 76 entries including 0. The user is responsible for making sure that the table size is large enough to handle the statement in question.

The jump table option is set using compiler option J (which MUST be the last option in an option list) and is disabled as soon as the first CASE statement has been encountered. Instead of using J+ and J-, the jump option takes one argument. The argument to the J option is the ordinal value of the last desired jump table entry. The above example, recoded to use a jump table would appear as follows:

```
{$J5 maximum value is violet, and the ordinal
     value of violet is five ( ord(violet) = 5 )
     so this will generate table entries for 0..5 }
CASE COLOR OF
    RED, YELLOW, BLUE: PRIMARY := TRUE;
    ELSE: PRIMARY := FALSE
    END;
```

If one of the CASE branches above had contained a CASE statement, it would not have been generated with a jump table unless there had been another J option used after the start of the first CASE statement.

When using the J option to create a CASE statement with a jump table, there is NO range checking done on the case selector.

SEPARATE COMPILATION

Pascal/Z versions 3.2 and later support separate compilation of user programs. Separate compilation is the ability to divide a large program into two or more pieces so that each piece can be compiled and assembled separately and then linked together. This feature is especially useful when modifying a large program since the time required to re-compile, assemble and link a 5,000 line program can sometimes reach one hour, while with separate compilation this can often be reduced to as little as ten minutes.

Now that a motivation for separate compilation has been established, we can proceed to the actual details of how it is and is not used.

1) Separate compilation in Pascal/Z assumes that there is a main module which contains the main program and external declarations (the same as those for an assembly language routine) for all external (i.e. those in another module) routines.

2) Each routine in a module which is not the main module must be declared as if it were being declared for the first time (i.e. it must have a fully defined formal parameter section and this must agree with the declaration in the main module, although the compiler will NOT check to make sure that this is the case).

3) Each separate module has access to anything declared at the global level in the main module, including externally declared routines in other separate modules.

4) Because of assembler and linker limitations all routines which are accessed from another module must have assembler legal names and have only eight significant characters. Names of externally declared procedures must not conflict with names already in use by MAIN.SRC and the library.

5) At any time a separate module may be changed, re-compiled, re-assembled and re-linked without any other changes. However, if there are any changes to the main module, then all other modules must be re-compiled, re-assembled and re-linked also -- this encourages the user to make the main module as small as possible to reduce the probability of its requiring change.

6) To use separate compilation no changes to the main module are required except those regarding declaration of externals as described above; however, when compiling the main module, the compiler must be invoked as follows:

PASCAL48 <filename>.xxxy

(where xxx are drive letters as described in HOW TO RUN PASCAL/Z and y is the drive on which to write two new files <filename>.SYM and <filename>.TYP). These files

contain symbol and type information used during compilation of the separate modules.

7) The syntax for separate modules differs from that of the main module as follows:

```
{Main program heading}
 Program <main filename>(0);

 External  <main  filename>::<module  name>(X);

<zero or more procedure/function declarations
 which need not have been declared in the main
 module (but if they were not declared in the
 main module they can NOT be accessed outside
 this module> .
```

The "X" in the external module heading represents the number of the module, and must be in the range 0 to 15. The default is 0. The numbering of modules is used when the compiler options T (trace) and/or E (extended error messages) are enabled, or when using InterPEST. Both the statement number and the module number (as specified by the "X" in the module heading) in which said statement is contained will be listed when the T and/or E options are enabled.

Note -- separate modules must end with a period '.' .

The compiler is invoked in a manner similar to that for the main module except that the fourth drive letter indicates the drive from which to read the two auxiliary files.

8) Separate modules are assembled with EMAIN.SRC rather than MAIN.SRC. This prevents the duplication of a few common routines and insures that the correct ENTRY and EXT statements are used. The main program module is still assembled with MAIN.SRC.

9) The error message 'premature EOF' may occur when using separate compilation. This generally means that the CP/M file names of the modules do not correspond to the internal module names as specified in the module headings. The problem is easily solved by checking to make certain the names correspond. This error message will also occur if the fourth drive letter is not specified when compiling the program modules.

This example of a main module and two separate modules should help clarify the use of separate compilation.

```
program test(0);   { This is the start of the file TEST.PAS }
const max_score = 100;
      min_score = 000;
type  score = min_score .. max_score;
      student = (dave,john,bill,peter,susan,mary,ruth,linda);
      test = ( math, geography, history, english, science );
var   blue_book: array[ student ] of array[ test ] of score;
      i,j,k: integer;
      s1, s2: student;
      t1, t2: test;

function average( name: student ): score; external;
function classavg: score; external;
procedure hilo( name: student; var high, low: score ); external;

begin
    for s1 := dave to linda do
     for t1 := math to science do begin
         write( s1:1, '''s ', t1:1, ' score is -- ' );
         readln( blue_book[ s1, t1 ] );
     end;
    for s1 := dave to linda do begin
     write( s1:1, '''s average score is: ', average( s1 ):3 );
     i := 0;
     j := 0;
     hilo( s1, i, j );
     writeln( ' with a high of ', i:3, ' and a low of ', j:3 );
    end;
    writeln;
    writeln( 'The class average is: ', classavg:3 )
end.


{ This is the start of the file INDIV.PAS }
External  test::indiv(1); { report on individual student }

function average( name: student ): score;
var i: integer;
    j: integer;
    t: test;
begin
    i := 0;
    j := 0;
    for t := math to science do begin
     i := i + 1;
     j := j + blue_book[ name, t ]
    end;
    average := j div i
end;

function min( i, j: integer ): integer;
begin
    min := i;
    if j < i then min := j
end;
```

```
function max( i, j: integer ): integer;
begin
    max := i;
    if j > i then max := j
end;

procedure hilo( name: student; var high, low: score );
var t: test;
begin
    low  := 100; { minimum score is <= 100 }
    high := 0;   { maximum score is >= 0   }
    for t := math to science do begin
     low  := min( low , blue_book[ name, t ] );
     high := max( high, blue_book[ name, t ] );
    end;
end;



{ This is the start of the file CLASS.PAS }
External    test::class(2);

function classavg: score;
var i,j: integer;
    s: student;
    t: test;
begin
    i := 0;
    j := 0;
    for s := dave to linda do
     for t := math to science do begin
         i := i + 1;
         j := j + blue_book[ s, t ]
     end;
    classavg := j div i
end;
```

This example has been tested and is KNOWN to work with Pascal/Z 4.0. The main module is called 'TEST' and the two subordinate modules are called 'INDIV' and 'CLASS' respectively. The following command sequence will compile, assemble and link this program:

```
pascal48 test.aaaa
asmbl main,test/rel
pascal48 indiv.aaaa
asmbl emain,indiv/rel
pascal48 class.aaaa
asmbl emain,class/rel
link /n:test test indiv class /g
```

The sample module 'INDIV' has two local functions, MIN and MAX, which can NOT be accessed anywhere outside of the INDIV module. Had it been desired, any of the routines in INDIV could have called CLASSAVG and CLASSAVG could have called HILO and AVERAGE.

PASCAL/Z EXTERNAL ROUTINES

EXTERNAL routines are used to communicate with the world outside of Pascal/Z. To Pascal/Z, external routines look and behave EXACTLY LIKE INTERNAL routines; so using an EXTERNAL routine is the same as using a regular Pascal procedure/function. EXTERNAL routines are declared just like FORWARD declared procedures and functions except that instead of using the reserved word FORWARD, the reserved word EXTERNAL is used instead. EXTERNAL modules should be assembled with EMAIN.SRC, while the main program should be assembled as usual with MAIN.SRC.

The following conventions must be followed in order for EXTERNAL routines to work properly:

1) The X, Y and alternate BC, DE, HL registers must be maintained.

2) Upon return from the EXTERNAL routine the accumulator must contain a zero.

3) If the routine is a function then:
   a) BOOLEANs return CARRY SET -> TRUE
                      CARRY CLEAR -> FALSE
   b) Other non-REAL scalars return value in DE register pair
   c) REALs return in the four bytes above the function parameters on the stack
   d) Structured types return in the n bytes above the parameter list on the stack (where n = size of return value)

4) Each EXTERNAL routine is responsible for removing all of its parameters from the stack before returning.

All parameters to EXTERNAL routines are passed on the stack. They are pushed onto the stack in the order that they are declared. The table in Appendix One shows exactly what the stack looks like after they have been pushed. After all of the parameters have been pushed onto the stack the EXTERNAL routine is called (using the Z-80 CALL instruction).

In the case of external procedures, variable values to be returned simply replace the variables passed; that is, the external routine accesses the address of the variable provided on the stack, uses this address to get the variable itself, processes it, and places the new value back at the same address. The external routine may alter value parameters passed directly on the stack, but this will have no effect on any variables in the program.

If we use the following contrived type declaration of COLOR, then the EXTERNAL function declared below it will return the color obtained by mixing two primary colors.

```
{ this type was specially designed so that the
  following is true:
    1) ord( red ) + ord( yellow )      = ord( orange )
    2) ord( red ) + ord( blue )      = ord( violet )
    3) ord( yellow ) + ord( blue )      = ord( green ) }
COLOR = ( CLEAR, RED, BLACK, BLUE, VIOLET, YELLOW,
          ORANGE, WHITE, GREEN );
        ....
FUNCTION MIX( PRIMARY1, PRIMARY2: COLOR ): COLOR; EXTERNAL;


; the assembly language routine will look like this
        ENTRY   MIX     ;entry point for the linker
MIX:    POP     H       ;get the return address
        POP     D       ;get the two colors
        MOV     A,D     ;get primary1 in A
        ADD     E       ;add in primary2
        MOV     E,A     ;set low half of return value
        XRA     A       ;clear accumulator
        MOV     D,A     ;set high half of return value
        PCHL            ;return
```

Note that the ENTRY point of the routine must be declared for the linker (only the first eight letters of the ENTRY point will be significant).

In the file EMAIN.SRC, which is included on the distribution disk, there are two macro definitions, ENTR and EXIT. These macros may be assembled with your EXTERNAL routines and used to simplify the writing of your routines. ENTR is used as follows:

```
    ENTR    D,LVL,VSIZ
```

where D is a dummy argument (that is, any value may be used), LVL is the declaration level of the routine (the global level is level 1, for EXTERNAL routines the value 2 should be used) and VSIZ is the number of bytes of local variables necessary for the EXTERNAL routine. Assuming n (where n is a nice number) bytes of parameters to the EXTERNAL routine then they may be addressed as

```
      n+7(IX) ;first byte of parameters  = 8+n-1(IX)
      n+6(IX) ;second byte of parameters = 8+n-2(IX)
        8(IX) ;last byte of parameters   = 8+n-n(IX)
```

Local storage (as requested by VSIZ) may be addressed as

```
        0(IX) ;first byte of local storage
  1-VSIZ(IX) ;last byte of local storage
```

And for non-REAL functions the return value must be stored in

```
        2(IX) ;low byte of return value
        3(IX) ;high byte of return value
```

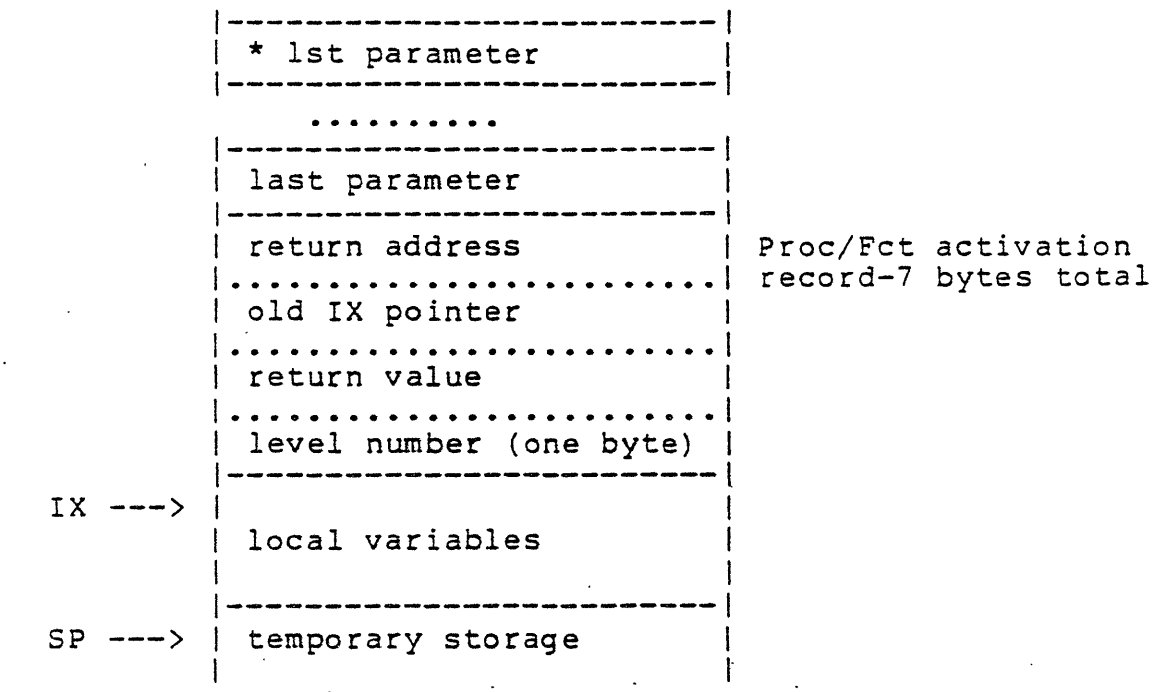
the return value is initialized to 0 by ENTR.

For real functions the return value must be stored in

```
       n+8(IX) ;least significant byte
          .  .  .
      n+11(IX) ;most significant byte (exponent)
```

For structured functions the return value must be stored in
```
n+8(IX) ; least significant byte
n+size-1(IX) ; most significant byte
```

A pictorial representation of the stack after execution of the ENTR macro is as follows:



EXIT is used as follows:

```
    EXIT   D,PARMSZ
```

where D is a dummy argument and PARMSZ is the number of bytes of parameters to the EXTERNAL routine. This routine will remove local storage and parameters, restore the stack and IX registers to their correct values, clear the accumulator, install the correct return value and return to the routine which called the EXTERNAL routine.

The EXTERNAL routine MIX can be recoded using ENTR and EXIT as follows:

```
; the assembly language routine will look like this
        ENTRY   MIX             ;for linker
PRIM1:   EQU     9              ;PRIMARY1 is 9 bytes above IX
PRIM2:   EQU     8              ;PRIMARY2 is 8 bytes above IX
RESULT:  EQU     2              ;low byte of result is 2(IX)
MIX:    ENTR    D,2,0           ;level 2, no local storage
        MOV     A,PRIM1(IX)     ;get primary1 in A
        ADD     PRIM2(IX)       ;add in primary2
        MOV     RESULT(IX),A    ;set low half of return value
;                               ;the high byte is already zero
        EXIT    D,2             ;two bytes of parameters
```

These last examples will allow a user program to do direct I/O.

```
PROCEDURE OUTPUT( PORT, VALUE: INTEGER ); EXTERNAL;

;
; assembly language output routine
;
PORT:   EQU    10            ;LOW BYTE OF PORT NUMBER
VALUE:  EQU     8            ;LOW BYTE OF OUTPUT VALUE
        ENTRY  OUTPUT        ;FOR LINKER
OUTPUT: ENTR   D,2,0        ;NO LOCAL STORAGE
        MOV    C,PORT(IX)    ;C <- OUTPUT PORT
        MOV    B,VALUE(IX)   ;B <- OUTPUT VALUE
        OUTP   B             ;OUTPUT THE VALUE TO THE PORT
        EXIT   D,4          ;DONE, FOUR BYTES OF PARAMETERS


TYPE BYTE = 0..255;
PROCEDURE INPUT( PORT: BYTE; VAR VALUE: BYTE ); EXTERNAL;

;
; assembly language input routine
;
PORT:   EQU    11            ;LOW BYTE OF PORT NUMBER
HADDR:  EQU    10            ;HIGH BYTE OF INPUT VALUE ADDR
LADDR:  EQU     9            ;LOW BYTE OF INPUT VALUE ADDR
LENGTH: EQU     8            ;LENGTH OF CALL-BY-REF INTEGER
        ENTRY  INPUT         ;FOR LINKER
INPUT:  ENTR   D,2,0         ;NO LOCAL STORAGE
        MOV    C,PORT(IX)    ;C <- INPUT PORT
        INP    B             ;INPUT THE VALUE FROM THE PORT
        MOV    H,HADDR(IX)   ;GET HIGH BYTE OF ADDRESS
        MOV    L,LADDR(IX)   ;GET LOW  BYTE OF ADDRESS
        MOV    A,LENGTH(IX)  ;GET SIZE
        CPI    2            ;CHECK FOR 2 BYTE INTEGER
        JRNZ   NOPE         ;NO, 1 BYTE INTEGER
        MVI    M,0          ;YES, CLEAR HIGH BYTE
        DCX    H            ;POINT TO LOW BYTE
NOPE:   MOV    M,B          ;STORE LOW (OR ONLY) BYTE
        EXIT   D,4          ;DONE, FOUR BYTES OF PARAMETERS
```

If a Pascal routine is to be used as an EXTERNAL routine for other Pascal programs and you do not wish to use separate compilation, then you must assemble it with EMAIN.SRC in order to create a .REL file which can be linked to the main Pascal program. Before assembling the EXTERNAL routine you must edit the .SRC file containing the Z-80 macro code for that routine.

You can do this as follows (this example is taken from the OVERLAY section later in the manual):

1) Add an ENTRY statement at the beginning of the .SRC file to identify the name of the routine(s) within the file for the linker.

```
     E.G.    ENTRY   MIN,MAX
```

where MIN and MAX are the names of procedures/functions within the module. (No spaces after the comma or the assembler will become very upset.)

2) Delete all EXTR and EXTD macro instructions from the .SRC file.

3) You must also identify each procedure/function for the linker by placing the name of the procedure/function, followed by a colon, immediately prior to the label indicating the beginning of that procedure/function. (A label which marks the beginning of a procedure or function will always be followed by an ENTR macro instruction. In the .SRC file, the code for the procedures/functions will be found in the same order in which the routines occur in the Pascal module.)

EXAMPLE:

```
MIN:
L140
     ENTR D,2,0
     STMT D,1
     MOV L,10(IX)
        .
        .
        .
L148
     EXIT D,4
     ENTRY L140
MAX:
L161
     ENTR D,2,0
     STMT D,4
     MOV L,10(IX)
        .
        .
        .
L169
     EXIT D,4
     ENTRY L161
L99
     ENTR D,1,0
     STMT D,7
     FINI
```

4) All code which may have been generated as part of the main program should be removed since you are only concerned with extracting one or more procedures or functions. To do this, delete all code from the L99 label to the end of the file.

5) Assemble and link as follows:

```
A>ASMBL EMAIN,MINMAX/REL
A>LINK /N:<com file name> <main program> MINMAX /G
```

OVERLAYING

Pascal/Z implements overlaying of modules containing procedures and functions, in order to allow programs to be executed which might not normally fit in available memory. An overlaying program is comprised of a resident program which remains in memory throughout the execution of the program, and one or more overlay modules, containing procedures and/or functions, which are swapped into an allocated section of memory prior to a call of a routine contained within an overlay module.

Overlaying is implemented in Pascal/Z as follows:

1) The program is broken into a main module and separate modules in the same manner as a separately compiled program (see section on SEPARATE COMPILATION). Modules which are to be overlay modules must contain routines which are accessed ONLY from the resident program or from other routines within the overlay module. Overlaying between overlay modules is NOT allowed. Other separate modules are allowed and become a part of the resident program.

2) To call a procedure or function in an overlay module, the Pascal/Z OVERLAY routine must first be called using the command:

```
OVERLAY(< name of file containing overlay module >);
```

followed by one or more calls to procedures or functions in the overlay module.

The < name of file containing overlay module > may be either a quoted string or an ARRAY[1..N] of CHAR. Note that OVERLAY is a predeclared identifier in Pascal/Z.

3) The main module and all separate modules, both overlay and non-overlay, are compiled and assembled in the same manner as a separately compiled program.

4) Before linking, the resulting .REL files must be processed by OVLYGEN.COM, a program included on the Pascal/Z distribution disk. OVLYGEN.COM generates memory maps of the main module (MAINMAP.REL) and the library (LIBMAP.REL), a memory map of the entry points of all routines in overlay modules (OVLYMAP.REL), allocates an area in memory where overlaying will take place, and generates an output file which contains the commands necessary to link the program and its overlay modules.

5) When the link commmands generated in step 3 are executed, the results are i) a .COM file which is the main program and all linked library modules, and ii) one or more files with no extension -- each containing an object code overlay module. When the program is run, these overlay files are loaded at the correct location in memory by the OVERLAY routine in the Pascal/Z run time library.

6) In order to minimize the size of the resident module, it is advisable to modify LIB.REL, the Pascal/Z run time support library provided on the distribution disk, prior to processing it using OVLYGEN.COM. This can be easily done using the Query option of the linker.

To determine which library modules should be included in the library for a specific overlayingl program, the following procedure may be followed:

> After all modules have been compiled and assembled, the main module should be linked with the standard LIB.REL, using the /S (search) option and the /V (verbose) option of the linker. A list of all modules linked will be generated.
>
> Then follow the same procedure with each separate module, overlay module, and external module.
>
> A composite list of all library modules linked during these 'pre-links' will determine those library modules which must be included in the program's library.
>
> Use the /Q (query) option to generate the new library by loading the standard library as the input file and excluding those modules not included in the composite list generated above. It is important that the relative position of modules within the library be maintained, and this is most easily achieved using the query option of the linker.

If the standard library is not modified prior to processing by OVLYGEN.COM, the .COM file generated after linking will contain all the library modules, whether or not they are needed, and will probably be much larger than necessary, perhaps defeating the purpose of overlaying.

The following example should help clarify overlaying and the use of OVLYGEN.COM. The program TEST is almost identical to the example in the section describing SEPARATE COMPILATION. But in this case, the program contains a main module, a separate module, a module containing two Pascal external functions, and two overlay modules.

```
{ Main module contained in file TEST.PAS }

program test(0);
const max_score =100;
      min_score =000;
type score = min_score .. max_score;
        student = ( dave,john,bill,peter,susan);
        test = (math,geography,history,english,science);
var     blue_book: array[ student] of array[ test ] of score;
        i,j,k:integer;
        s1,s2:student;
        t1,t2:test;
```

```
function average( name:student): score; external;
function classavg: score; external;
function min( i, j: integer ): integer; external;
function max( i, j: integer ): integer; external;
procedure hilo( name: student; var high,low:score ); external;

begin
        for sl := dave to susan do
                for tl := math to science do begin
                  write( sl:1, '''s ', tl:1, ' score is -- ' );
                  readln( blue_book[ sl,tl ] );
                  end;
        overlay( 'indiv' );             { first overlay }
        for sl := dave to susan do begin
                write( sl:1, '''s average score is : ',
                                        average( sl ):3 );
                i := 0;
                j := 0;
                hilo(sl, i, j );
                writeln( ' with a high of ', i:3,
                                ' and a low of ', j:3 );
                end;
        writeln;
        overlay( 'class' );             { second overlay }
        writeln( 'The class average is : ', classavg:3 )
end.


{ separate module contained in file AVE.PAS }

external test::ave;

function average( name: student ): score;
var i, j: integer;
        t: test;
begin
    i := 0;
    j := 0;
    for t := math to science do begin
        i := i + 1;
        j := j + blue_book[ name, t ]
        end;
    average := j div i
end;



{ external functions contained in file MINMAX.PAS }
{ see section on Pascal/Z External Routines       }

program minmax;

function min( i, j: integer ) : integer;
begin
  min := i;
  if j < i then min := j
end;
```

```
function max( i,j:integer ): integer;
begin
    max := i;
    if j > i then max := j
end;

begin
end.


{ overlay module contained in file INDIV.PAS }

external test::indiv;

procedure hilo( name: student; var high, low: score );
var t: test;
begin
    low := 100;
    high := 0;
    for t := math to science do begin
        low := min( low, blue_book[ name, t ] );
        high := max( high, blue_book[ name, t ] );
        end;
end;



{ overlay module contained in file CLASS.PAS }

external test::class;

function classavg: score;
var i,j:integer;
    s: student;
    t: test;
begin
    i := 0;
    j := 0;
    for s := dave to susan do
        for t := math to science do begin
            i := i + 1;
            j := j + blue_book[ s,t ]
            end;
    classavg := j div i
end;
```

The following command sequence will compile and assemble the modules of this program:

```
pascal48 test.aaaa
asmbl main,test/rel
pascal48 ave.aaaa
asmbl emain,ave/rel
pascal48 indiv.aaaa
asmbl emain,indiv/rel
```

```
pascal48 class.aaaa
asmbl emain,class/rel
pascal48 minmax.aaa
 { see section on Pascal/Z external routines }
 { for modifications to MINMAX.SRC prior to  }
 { assembly                                  }
asmbl emain,minmax/rel
```

To generate a new library do:

```
link test /v lib/s /r /e
link ave /v lib/s /r /e
link minmax /v lib/s /r /e
link indiv /v lib/s /r /e
link class /v lib/s /r /e
```

After each link note the modules that have been linked and then generate a new library using the command:

```
link /l:newlib lib/q /e
```

OVLYGEN.COM is then executed, prompting the user for:

1) the name of the .REL file containing the Pascal/Z CHAIN routine if chaining is being done between this program and another program.

2) the name of the .REL file containing the main module.

3) the name of all .REL files containing separate modules.

4) the name of all .REL files containing external routines.

5) the name of all user-created libraries.

6) the name of all .REL files containing overlay modules.

7) the name of the Pascal/Z run time support library to be used.

(If using the debugger InterPEST with an overlaying program, the debugger should be specified as a user-created library.)

At the completion of execution, OVLYGEN.COM will generate a file TEST.SUB which will contain the following commands.

```
link /n:test test ave minmax ovlymap newlib /u /e
link /o:0675 /n:indiv indiv mainprog libmap /u /e
link /o:0675 /n:class class mainprog libmap /u /e
```

Note: A '/u' is generated in each linker command line. This is included to flag any library modules which might have been inadvertently left out of the modified library. If any library entry points are listed as unresolved external symbols, the modified library should be regenerated, even if the '/e' option (which loads the library - usually LIB.REL) resolves the indicated unresolved externals. Not doing so will result in unpredictable, and often disastrous results.

After execution of these commands, TEST.COM may be run.

INCLUDE FILES

Pascal/Z provides an INCLUDE facility, with which any file may be included at any point in a Pascal/Z program.

The file or files to be INCLUDEd must be on the default drive or the drive must be specified, followed by a colon and the filename. (The compiler will not check to make certain the file is present, and will generate no error message if it is not.)

To use INCLUDE files, type the following at the point at which the file is to be inserted:

```
{$I<filename> }
```

The "I" specifies that the file following it should be INCLUDEd. Please note that $I+ or $I- in a comment with no filename specified indicates that the IMBED compiler option should be enabled/disabled -- be careful that there are no spaces or characters between the "I" and the filename. Also, there must be a space between the filename and the closing bracket (}).

Care must be taken to ensure that there are no conflicting declarations in the various files -- it is very easy to INCLUDE a file which uses variables already declared in another of the files.

To INCLUDE more than one file in the same place, each filename must be specified in a separate comment.

INCLUDE files may be nested, limited only by available stack space.

CHAINING

While we advise against overlays and chaining, we realize that in some instances there is no choice. Overlays were discussed in an earlier section. In Pascal/Z there is also a way for one program to chain to another with the same global declarations (programs must chain at the main program level). In the Pascal program the following is used to do the chain:

```
FTXTIN( <name of file to call> ); CHAIN;
```

The <name of file to call> may be either a quoted string or an ARRAY[1..N] of char. Then each of the programs involved in the chaining must be re-linked with the chain module (shipped in source on the library diskette) as follows:

```
LINK CHAIN <Pascal REL filename>/N:<COM filename>/E
```

For CHAIN to work correctly it is imperative that it be the first file linked.

It is also possible for one program to chain to another with different global data areas by changing the ENTR macro as indicated in CMAIN.SRC.

If chained programs are to use the heap then the LAST routine in the library should be changed to set LAST to the size of the largest program being chained.

PASCAL/Z POINTERS

Pointers, combined with their associated intrinsic routines, NEW, MARK, and RELEASE, are the means by which a user can dynamically allocate and deallocate variable storage. While the PASCAL REPORT describes NEW and DISPOSE, we have implemented NEW, MARK and RELEASE. These procedures are used as follows:

NEW(P) P may be any pointer variable. NEW allocates storage for P^ from the heap and assigns P the address of this NEWly allocated storage.

MARK(P) P must be a pointer variable. MARK sets P equal to the present top of heap (the information stored in P is used by a subsequent RELEASE).

RELEASE(P) P is a pointer variable which has been previously "MARKed". RELEASE (P) releases all of the heap which has been allocated since P was "MARKed".

The RELEASE of a pointer to a file variable will close the output file pointed to.

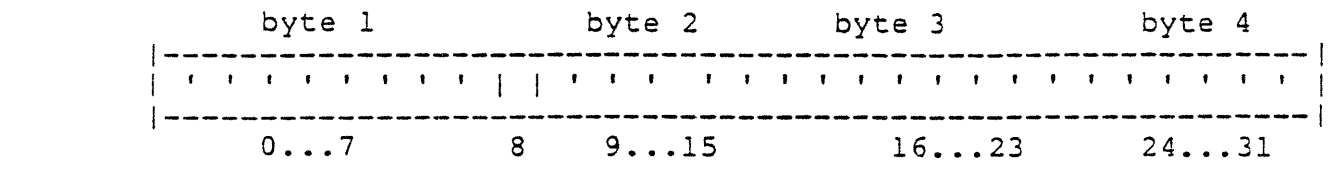
Note: A variable which has been "MARKed" should not be used for anything other than a RELEASE if you want to maintain the "MARK".

The program on the next page will input a list of names, store them in a linked list, output the list, release the storage and end.

```
PROGRAM POINTERS;
TYPE LINK = ^NAMEREC;
     NAMEREC = RECORD
               NAME: STRING 20;
               NEXT: LINK
               END;
VAR M1: LINK;        { FOR STORING THE MARK }
    FIRST: LINK;     { FOR FINDING THE FIRST NAME }
    LAST: LINK;      { FOR FINDING THE LAST  NAME }
    X: LINK;         { FOR CHASING THROUGH THE LIST }
BEGIN
    MARK( M1 );
    FIRST := NIL;
    REPEAT
        IF FIRST = NIL THEN BEGIN
            NEW( LAST );  { ALLOCATE A NEW RECORD }
            FIRST := LAST
        END
        ELSE BEGIN
            NEW( LAST^.NEXT );
            LAST := LAST^.NEXT
        END;
        WRITE( 'Name ( * = DONE ) ' );
        READLN( LAST^.NAME );  { GET PERSONS NAME }
        LAST^.NEXT := NIL
    UNTIL LAST^.NAME = '*';
    { PRINT OUT THE NAMES }
    X := FIRST;
    WHILE X^.NEXT <> NIL DO BEGIN
        WRITELN( X^.NAME );
        X := X^.NEXT
    END;
    { RESTORE THE STORAGE }
    RELEASE( M1 );
END.
```

PASCAL/Z FLOATING POINT NUMBERS

In Pascal/Z all floating point numbers are 4 bytes and are organized as follows:



There is an implied binary point to the left of bit 9. The value of the mantissa is multiplied by 2^(exponent) to achieve the final value.

Floating point precision is approximately 6 1/2 decimal digits (there will be a roundoff error in the seventh digit). If this is inadequate for your purposes, the fixed point routines described in Appendix Three might provide sufficient precision (or try Pascal/BZ, the business version of Pascal/Z).

Other floating point formats are possible (i.e. AMD9511 could be used to process floating point); contact InterSystems for details if you need to use another format.

FORMATTING OUTPUT

When outputting the data types INTEGER, REAL and BOOLEAN, it is often desirable to format the output into columns. This is done by specifying a field width such that all data is properly aligned.

In Pascal/Z, the standard field width for outputting the data types INTEGER, REAL and BOOLEAN is eight. This can be overridden by specifying the minimum field width preceded by a colon, as follows:

```
X := LAURIE
WRITE( X : 10 );
```

X will be written in ten places and aligned to the right:

```
bbbbLAURIE
```

where b indicates a blank place.

If the item to be output requires more than the allotted number of columns, it will use as many as is necessary.

When outputting a REAL, the user may specify not only the field width, but also the number of digits to be printed after the decimal point. This is done by adding another colon followed by an integer specifying the number of digits to be placed after the decimal point.

Jensen & Wirth dictate that floating point output should always be preceded by two spaces--one blank and one for the sign (which is only actually printed if it is negative; otherwise it is left blank).

Example:

```
X := 12.352
WRITE( X : 7 : 2 );
```

This will output X as follows:

```
bb12.35
```

The number will be rounded to display the prescribed number of places after the decimal point. If the number of places specified to be output is larger than the number of existing digits, it will be padded with zeroes.

With the same number as a negative (i.e. -12.352), the following statements will return the following results:

```
WRITE( X : 7 : 2 ) -----> b-12.35

WRITE( X : 5 : 2 ) -----> b-12.35
```

Note that although in the second example a field width of five is specified, since the two preceding spaces are needed, the field

width is automatically expanded to accept the number. Whenever the field width specified is not large enough to output the number as specified, as many places as necessary will be taken.

Also note that the number is printed in fixed point notation, rather than exponential/scientific notation. Had X originally been specified in scientific notation, the result would be as follows:

```
X := 1.34E+3
WRITE( X : 6 : 4 );
```

Result: bb1340.0000

If the number is too large to be represented in fixed point notation (e.g. 1.34E+25), the number will be output in scientific notation, but will still place the proper number of digits after the decimal point.

```
X := 1.34E+25;
WRITE( X : 6 : 4);
```

Result: b1.3400E+25

Note as well that the decimal point occupies one place, and must be accounted for when specifying field width.

ASSEMBLER AND LINKER ERRORS

There are a few assembler errors which can occur during the assembly of a Pascal/Z program. These result from the few program errors which are not caught by the compiler. The first error is an attempted invocation of a FORWARD declared procedure/function which is never actually defined; in this case the error will be:

```
Symbol not found
       CALL     L?????
```

The second error is a GOTO to a label which was declared but not defined; in this case the error will be:

```
Symbol not found
       JMP      L?????
```

The next error is a READ/WRITE to/from a textfile of a parameter which is larger than 255 bytes; in this case the error will be:

```
Argument too big
       MVI      <register>,<value>
```

MEMORY USAGE

This section is included to allow users of Pascal/Z to make efficient use of memory by describing the amount of storage necessary for certain types of variables and statements. Since in Pascal/Z the reserved word PACKED is implicit in all ARRAY declarations its use is unnecessary (but still allowed).

BOOLEANs
stored 1 byte/variable

CHARs
stored 1 byte/variable

INTEGERs
stored 2 bytes/variable

REALs
stored 4 bytes/variable

Enumeration types
stored 1 byte/variable

Subranges
stored in one byte unless the base type is INTEGER and the range doesn't fit in one byte (the lower bound is less than zero or the upper bound is greater than 255).

Pointers
stored 2 byte/pointer

File variables
stored 300 bytes/variable

ARRAYs & RECORDs
require the amount of memory equal to the sum of the requirements of the individual elements. Variant RECORDs require storage sufficient to store the largest variant case.

Quoted strings
each quoted string takes up three bytes plus one additional byte for each character in the string.

Declared constants
numerical constants require no memory; string constants use one byte/character plus one byte of overhead.

Procedure activation (each call of a procedure)
each activation record requires seven bytes of storage plus the memory for the parameters and local variables. This storage is allocated dynamically from the run-time stack and is released when the procedure is exitted.

Function activation
same as procedure activation, except for REAL functions which require eleven bytes of memory.

Value parameters
each parameter requires storage as defined above.

Reference parameters (VAR parameters)
all reference parameters require two bytes of storage except for those of base type INTEGER which require three bytes.

SETs
All sets are stored in 32 bytes.

FOR statements
require four bytes of storage for each active FOR loop.

WITH statements
require two bytes of memory for each active dynamic WITH statement.

All of the memory usage described in this section (with the exception of quoted strings and string constants) is allocated dynamically from the run-time stack and is only used when a particular routine or statement is active.

STACK AND HEAP ORGANIZATION

This section is intended mainly for those users interested in using Pascal/Z programs in a multi-tasking environment.

ALL code generated by the Pascal/Z is completely ROM-able and is also re-entrant as long as separate stacks and heaps are maintained as described in this section.

All Pascal/Z programs start by initializing their stacks. The code to do this is in MAIN.SRC. ALL non-dynamic variables are stored on the run-time stack. By changing the stack initialization code, many different processes can run with the same memory image of the Pascal/Z program as long as each one has a different, and non-overlapping, stack.

At any given time during program execution the IY register points to the global variable stack frame and the IX register points to the local stack frame.

Although all variables are stored on and accessed through the stack, the stack can not be relocated once program execution has begun. This is because there are often references from one part of the stack to another part of the stack which, in the interest of efficiency, are stored as absolute addresses and not stack relative addresses.

The heap, like the stack, is allocated at run-time, not compile time. This means that the heap too may be initialized at a different point in memory for each process. The initial value of the heap is determined by the LAST module; this could easily be changed whenever desired. It is initially set to the first location following the user program.

The run-time package is set up to allocate the stack from the top of memory downwards and the heap from the bottom of memory upwards and to give an appropriate error if they collide. If you are allocating many stack-heap pairs for different processes then it is most reasonable to allocate them in contiguous memory to ensure that the stack and heap overflow checking is not defeated.

INSTALLING PASCAL/Z PROGRAMS IN ROM

When installing a Pascal/Z program into a ROM it is NOT necessary to make any changes to the program. However, it is possible to further reduce the size of the object code before burning a ROM. This is especially true for programs which are to run in a dedicated environment.

When your program is assembled and linked, a certain amount of error recovery and termination code is included. This includes code to close any open output files, to print error messages and to check for ^C. If your program does not use any files for output, then much of this code can be eliminated by replacing the

```
JMP  L0
```

instruction in the FINI macro (which is found in MAIN.SRC) with a

```
JMP <location at which execution should resume
      after completion of the program>
```

by replacing the

```
JZ  ERROR
```

instruction in the CTRL macro (in MAIN.SRC) with the same JMP instruction as above, and by removing the

```
CNZ  CLSOT
```

in the ENTEXT.SRC module.

Similarly, if you do I/O but do not do any floating point operations, the floating point code can be eliminated by modifying the INPT.SRC module with the removal of the

```
CZ  FLTIN
```

call.

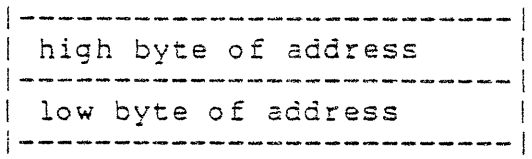
APPENDIX ONE

PARAMETER STACK CONFIGURATIONS

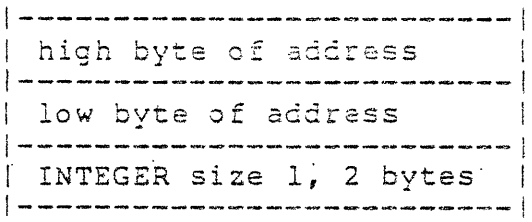
This appendix describes the form of parameters that are pushed onto the stack for use in external routines.

In this appendix all diagrams have the high memory locations towards the top of the page and the lower locations towards the bottom of the page. In addition each box represents one byte unless otherwise specified.

All reference parameters (except those of base type INTEGER) take the form of a two byte address and are passed like this:



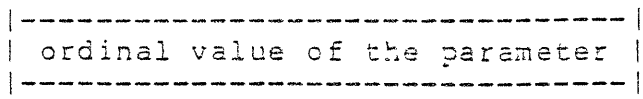
Call by reference INTEGERs take form:



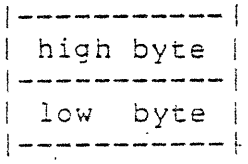
All reference parameter addresses (except those of type FILE) are the address of the highest byte (highest memory location) of the variable. FILE parameter addresses are the lowest byte of the variable since operating systems are usually concerned with the lower end of the buffer.

Value parameters are copied directly onto the stack and take the form described below:

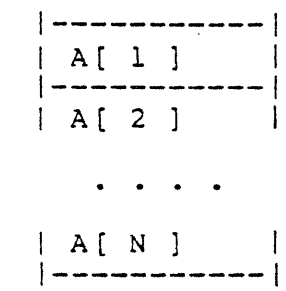
BOOLEANs, CHARs, enumeration types, one byte INTEGERs:



Two byte INTEGERs:

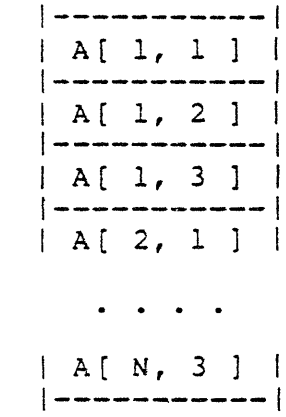


ARRAYs:



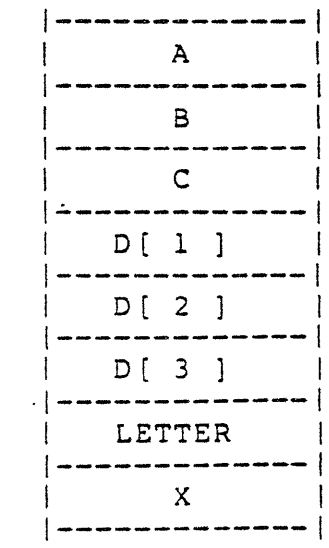

```
ARRAY[ 1..N ] OF COLOR
```

Multi-dimensional ARRAYs:



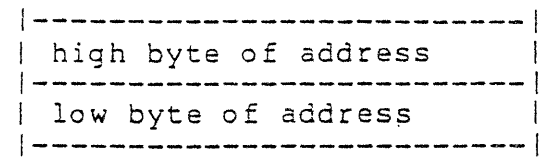

```
ARRAY[ 1..N ] OF
  ARRAY[ 1..3 ] OF
    BOOLEAN;
```

RECORDs:

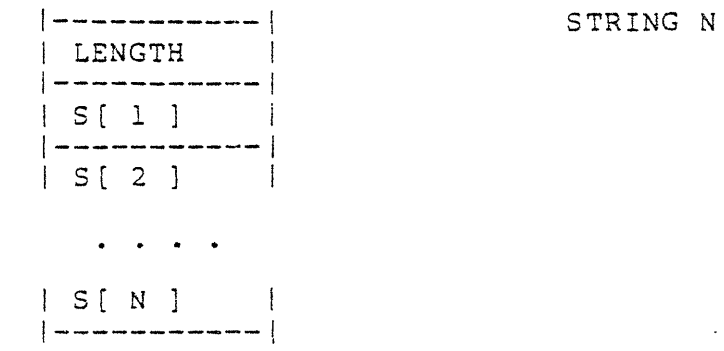



```
RECORD
     A, B, C: 0..255;

     D: ARRAY[ 1..3 ] OF
          BOOLEAN;

     LETTER: CHAR;

     X: BOOLEAN
END;
```

Pointer variables:

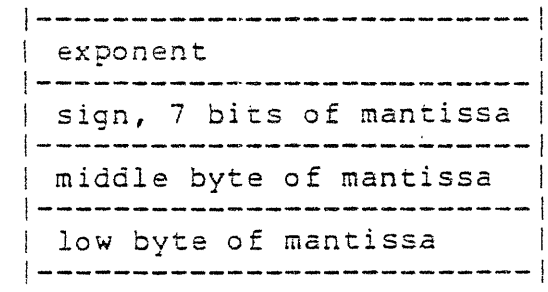


STRINGs:

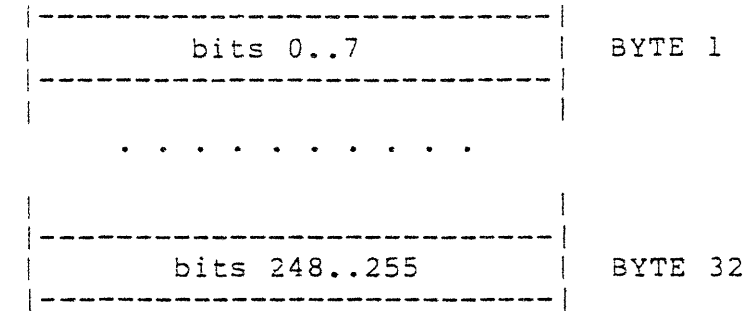


The entire string is passed even if the length is zero.

REALs (floating point numbers):



SETs:



APPENDIX TWO

TROUBLESHOOTING

Most problems encountered when using the Pascal/Z compiler are manifestations of hardware problems.

It is common for memory boards, in non-IEEE S100 standard bus machines, to run memory tests for many hours and then not be able to run the Pascal compiler. This is usually due to the inability of the memory to be able to handle M1 states in all of the first 40K.

Often Pascal/Z will be the first program you ever run that uses all of high memory. Therefore it is in your best interest to save time and frustration by making sure your memory is good (this can sometimes be done by re-arranging memory boards).

Another problem is not enough memory. The compiler will output a message to this effect if this is the case (but will not be able to in the cases where the shortage is too severe).

If you are unable to read the disk, which is a single density standard CP/M diskette, then check the calibration of your drives.

If you receive the message 'Unable to Chain' then the problem is that PAS248 (or PAS254) is not on the currently logged-in drive.

If you receive the message 'Unable to overlay' then the problem is that one or more of the overlay modules called is not on the currently logged-in drive. This will occur if the modules DECS and PFSTAT are not on the drive with PASCAL48.COM and PAS248.

If you have performed the above steps and still can not successfully compile even the demo programs then contact the dealer from whom the Pascal/Z package was purchased or the factory for assistance.

APPENDIX THREE

FIXED POINT PACKAGE

The Fixed Point Package is a collection of procedures which perform arbitrary precision arithmetic in signed fixed-point decimal. The Pascal/Z fixed point routines are implemented in binary-coded decimal (BCD), and are packed two digits per byte. In addition to the four basic functions add (ADD), subtract (SUB), multiply ($MULT), and divide ($DIVD), functions are supplied that convert between real and fixed point and between string and fixed point.

The source code for these functions is supplied so that the user may include it in his or her program. The bulk of the code is in the file 'FIXED.PAS'. This has all of the procedures and none of the declarations. There are four constants, seven types, and two variables that must be declared. These are in the files 'FIXCONST.PAS', 'FIXTYPE.PAS', and 'FIXVAR.PAS' respectively. All three of these files are very short. In the event that any of the global identifiers used by the fixed-point package are the same as ones used by a user program, either the fixed-point code will have to be changed or the user code will have to be changed. We suggest that the user be aware of these identifiers when writing code that uses the fixed-point package.

Often the user will not need all of the functions supplied. In the interest of faster compilations, shorter listings, and smaller source and object files, the unused functions may be deleted as long as they are not called by functions that remain. The dependencies are as follows: subtract calls add, greater calls subtract, divide calls subtract, add calls subbyte and addbyte, just about everything calls shiftleft and shiftright.

The Type 'FIXED'

The user declares a fixed-point number as follows:

```
MYVAR: FIXED;
```

A fixed point number can be an element of an array, record or file. For example:

```
DIV_TOTALS: ARRAY[ 1..DIV_MAX ] OF FIXED;

DIV_INFO: RECORD
            NAME: STRING 40;
            DIV_LOCATION: 1..DIV_MAX;
            DIV_SALES: FIXED
          END;

MYFILE: FILE OF FIXED;
```

The type fixed takes up enough space to store the number of digits requested by the constants 'LEFT' and 'RIGHT' in a packed form plus one (for the sign).

The Constants 'LEFT' and 'RIGHT'

These constants are found in the file FIXCONST.PAS and are set by the user for the amount of precision needed. 'LEFT' specifies the number of decimal digits that fixed-point numbers are to have to the left of the decimal point. Likewise, 'RIGHT' specifies the number of digits to the right of the decimal point. Both numbers must be non-negative and their sum must be positive. A program may have only one size of fixed-point number. Also, a file written with one size of fixed-point cannot be read as another size. So, if you have a collection of programs that share common files with fixed-point numbers in them, they all must be the same size.

The Arithmetic Functions

Add, subtract, multiply, and divide all take two fixed parameters by value and return a fixed. Subtract takes the minuend as the first parameter and the subtrahend as the second parameter (the second is subtracted from the first). Divide takes the first parameter as the dividend and the second as the divisor (the first is divided by the second). All four functions will set the global boolean 'FIXEDERROR' true if there is an arithmetic overflow. Divide by zero will cause the flag to be set true and a value of zero will be returned.

All numbers to be operated upon must be read in as strings or as real numbers. Before the numbers can be used, they must be converted to fixed point numbers, using either the STRTOFIX (string to fixed) or REALTOFIX (real to fixed) routines described below. They can then be operated upon using the fixed point operators ADD, SUB, $MULT and $DIVD, or the relational operator GREATER, as shown in the example program. ONLY the fixed point operators may be used on fixed point numbers.

Once all arithmetical operations have been completed, the fixed point numbers must be converted to the format in which they were read in before they can be written out, using either the FIXTOSTR (fixed to string) or FIXTOREAL (fixed to real) routines.

Real and Fixed Conversions

The functions 'REALTOFIX' and 'FIXTOREAL' convert real numbers to fixed-point and vice-versa respectively. Both have their single parameters passed by value. 'FIXEDERROR' will be set if the number being converted is too big for the result type.

String and Fixed Conversions

The functions 'STRTOFIX' and 'FIXTOSTR' convert strings to fixed-point and vice-versa respectively. Both have their parameters passed by value.

STRTOFIX scans the string from left to right ignoring all characters other than the decimal digits, the minus sign, and the period (decimal point). Thus the string may have a dollar sign

and commas and it will still be converted properly (i.e. the strings '$12,314.43' and '12314.43' would be converted to the same fixed-point number).

FIXTOSTR takes three parameters. The first is the fixed number to be converted. The second is the format mode. The format mode is an enumerated type that specifies the operations to be performed on the number as it is being converted. The different formats follow:

none: No formatting is done, no zeros are suppressed.

suplzer: Leading zeros are suppressed.

supltzer: Both leading and trailing zeros are suppressed.

wdollar: Leading zeros are suppressed and a dollar sign is placed before the most significant digit.

wcomma: Leading zeros are suppressed and commas are inserted between every third digit to the left of the decimal point.

wboth: Leading zeros are suppressed, commas are inserted every between every third digit to the left of the decimal point, and a dollar sign is placed before the most significant digit.

The third parameter specifies the number of trailing digits to display past the decimal point.

Relational Function

The function 'GREATER' compares two fixed-point numbers. It returns true if the first operand is greater than or equal to the second operand.

USING THE FIXED POINT ROUTINES

The fixed point routines, including the constant, type and variable declarations, must actually be inserted into the Pascal/Z program; they are not part of the library LIB.REL.

This may be done using INCLUDE files (see page 79) as follows.

The user must first examine the file FIXCONST.PAS, which contains the constant declarations for the fixed point routines. He specifies the precision of the fixed point number in FIXCONST.PAS by setting the value of the constants LEFT and RIGHT to the precision desired. Once the precision is established, it CANNOT be changed within the same program to save storage space.

He must insert the declarations for the fixed point routines with the other declarations at the beginning of his program. The fixed point declarations must be included in the appropriate declaration

section before any declarations of fixed point numbers are made (otherwise the error "Identifier not declared" will be encountered). He can do this using INCLUDE files by typing

{$IFIXCONST.PAS } at the end of the constant declarations,

{$IFIXTYPE.PAS } at the end of the type declarations, and

{$IFIXVAR.PAS } at the end of the variable declarations.

Note, however, that the reserved words CONST, TYPE and VAR are contained in the fixed point declarations, and that these must be removed if any constant, type or variable declarations have already been made in the program. Failure to do this can cause disastrous results during compilation.

Then he may INCLUDE the fixed point routines in his program at any point before they are accessed (we suggest at the beginning of the procedure and function declarations) by typing:

```
{$IFIXED.PAS }
```

The entire Pascal/Z program is then compiled, assembled and linked as usual.

EXAMPLE

```
Program checkbook;

{ This is a simple checkbook balancing program designed }
{ to demonstrate the use of the fixed point package     }

const     max_t = 100;

{ The following comment INCLUDES the fixed point constant }
{ declarations.  The FIXCONST.PAS file has already been   }
{ editted to remove the reserved word CONST.              }

{$ifixconst.pas }
type

{ The following comment INCLUDES the fixed point type  }
{ declarations.  The FIXTYPE.PAS file has already been }
{ editted to remove the reserved word TYPE.            }

 {$ifixtype.pas }

     transaction = ( inq, dep, chk, stmt, stop );
     daterec = record
               month : ( jan, feb, mar, apr, may, jun, jul,
                     aug, sep, oct, nov, dec );
               day : 1..31;
               year : integer;
             end;
     transrec = record
                 oldbal, newbal : fixed;
                 date : daterec;
```

```
            t_type : transaction;
            amount : fixed;
          end;
    sequence = 1..max_t;


var

{ The following comment INCLUDES the fixed point variable }
{ declarations.  The FIXVAR.PAS file has already been     }
{ editted to remove the reserved word VAR.                }

{$ifixvar.pas }

    number : string 20;          { number to be read }
       dummy,                    { dummy variable for comparison }
    income,                      { present deposit }
    withdrawal,                  { present withdrawal }
    balance : fixed;             { current balance }
    option : transaction;        { current operation }
    done : boolean;              { test for stop }
    today : daterec;             { date }

       { array to store individual transactions }
    history : array[ 1..max_t ] of transrec;
    t# : sequence;             { transaction number }
    i : sequence;              { index variable }


{ forward declaration of procedure print }
procedure print; forward;

{$1-}  { turn off listing to eliminate fixed point }

{ The following comment INCLUDES the fixed point routines. }
{$ifixed.pas }

{$1+}  { turn the listing back on }

{ enter the date }
procedure getdate;
begin
  write( 'Enter date (E.G. JUN 17 1981) -- ' );
  with today do
    read( month, day, year );
end;

{ add deposits }
procedure increment;
begin
  write( 'Deposit -- ' );
  read ( number );
  { convert the string to a fixed point # }
  income := strtofix( number );
  { perform a fixed point addition }
  balance := add( balance, income );
  { store the current information in the array }
  with history[ t# ] do
```

```
  begin
    { perform a fixed point subtraction }
    oldbal := sub( balance, income );
    newbal := balance;
    date := today;
    amount := income;
    t_type := option;
  end;
    { convert the fixed point # to a string and write it out }
  writeln( fixtostr( balance, wboth, 2 ) );
  t# := t# + 1;
end;

{ decrement withdrawals }
procedure decrement;
begin
  write( 'Withdrawal amount -- ' );
  read( number );
  { convert the string to a fixed point # }
  withdrawal := strtofix( number );
  with history[ t# ] do
  begin
    oldbal := balance;
    newbal := sub( balance, withdrawal );
    date := today;
    t_type := option;
    amount := withdrawal;
  end;
  t# := t# + 1;
  { perform a fixed point subtraction }
  balance := sub( balance, withdrawal );
  history[ t# - 1 ].newbal := balance;
  { convert the fixed point # to a string and write it out }
  writeln( fixtostr( balance, wboth, 2 ) );
  { routine to check for overdrawn account }
  dummy := strtofix( '0' );
  if greater( dummy, balance ) then writeln( 'Oops! Overdrawn' )
end;

{ procedure to print out statement }
procedure print;
begin
  with history[ i ], date do
  begin
    { convert all fixed point #s to strings and write out }
    write( fixtostr( newbal, wboth, 2 ) );
    write( fixtostr( oldbal, wboth, 2 ) );
    write( month, day, year, t_type );
    writeln( fixtostr( amount, wboth, 2 ) )
  end;
end;

begin          { main program }
  done := false;
  t# := 1;
  getdate;
  write( 'Starting balance -- ' );
  read( number );
```

```
  { convert the string to a fixed point # and assign it to balance }
  balance := strtofix( number );
  repeat
    write( 'Option ( inq, dep, chk, stmt, stop ) -- ' );
    read( option );
    case option of
      inq: writeln( fixtostr( balance, wboth, 2 ) );
      dep : increment;
      chk : decrement;
      stmt : for i := 1 to t# - 1 do print;
      stop : done := true
    end;
  until done;
end.
```

The fixed point declarations and routines may also be inserted by typing or editing the files FIXCONST.PAS, FIXTYPE.PAS, FIXVAR.PAS and FIXED.PAS into the Pascal program itself.

(Note that the Fixed Point Package is supplied with Pascal/Z only, and that the routines are not intrinsic to the compiler. For users desiring greater speed, precision and flexibility, Pascal/BZ should be used. You will have received Pascal/BZ only if it was specifically ordered, as it is a distinct product from Pascal/Z. See Appendix Seven for details on Pascal/BZ.)

APPENDIX FOUR

PASCAL/Z USER'S GROUP

The Z Users' Group was formed by Charles Foster of Sacramento, California to provide a forum for information and discussion about Pascal/Z.

The organization publishes a bi-monthly newsletter containing information on current and forthcoming versions of Pascal/Z, as well as user comments and suggestions. The newsletter also describes public domain software, written in Pascal/Z, available from the Users' Group. There are currently twelve disks containing programs donated by users of Pascal/Z, with more to follow.

The cost to be placed on the mailing list for the newsletters is $9.00 (U.S.) annually. The cost for the User Disks is $10.00 per disk/volume. (ALL DISKS ARE CP/M COMPATIBLE, SOFT-SECTORED, SINGLE DENSITY, IBM 3740 FORMAT.)

Any request for further information must be accompanied by a self-addressed, stamped envelope. Contact:

Mr. Charles Foster
Z Users' Group
7962 Center Parkway
Sacramento, California
95823

916-392-2789 5PM - 10PM (PST)
916-447 6077 8AM - 5PM (PST)

APPENDIX FIVE

ITHACA INTERSYSTEMS LIMITED WARRANTY

ITHACA INTERSYSTEMS disclaims any warranty as to this product. This product is sold for commercial, and not consumer, use.

> SELLER MAKES NO WARRANTY EXPRESS OR IMPLIED, AND ANY IMPLIED WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE WHICH EXCEEDS THE FOREGOING WARRANTY IS HEREBY DISCLAIMED BY SELLER AND EXCLUDED FROM ANY AGREEMENT.

Buyer expressly waives its rights to any consequential damages, loss or expense arising in connection with the use of or the inabiity to use its goods for any purpose whatsoever.

No warranty shall be applicable to any damages arising out of any act of the Buyer, his employees, agents, patrons or other persons.

The remedies set forth herein are exclusive and the liability of Seller to any contract or sale or anything done in connection therewith, whether in contract, in tort, under any warranty, or otherwise, shall not, except as expressly provided herein, exceed the price of the equipment or part on which said liability is based.

No employee or representative of Seller is authorized to change this warranty in any way or grant any other guarantee or warranty.

APPENDIX SIX

ERROR MESSAGES

The following are the error messages listed by Jensen & Wirth on pages 119 - 121 of the USER MANUAL AND REPORT shipped as part of the Pascal/Z software package. The error messages which are specific to Pascal/Z are listed under the 398 category: implementation restriction.

Note that not all of the error messages are applicable to Pascal/Z.

1	error in simple type
2	identifier expected
3	'program' expected
4	')' expected
5	':' expected
6	illegal symbol
7	error in parameter list
8	'of' expected
9	'(' expected
10	error in type
11	'[' expected
12	']' expected
13	'end' expected
14	';' expected
15	integer expected
16	'=' expected
17	'begin' expected
18	error in declaration part
19	error in field-list
20	',' expected
21	'*' expected
50	error in constant
51	':=' expected
52	'then' expected
53	'until' expected
54	'do' expected
55	'to'/'downto' expected
56	'if' expected
57	'file' expected
58	error in factor
59	error in variable
101	identifier declared twice
102	low bound exceeds high bound
103	identifier is not of appropriate class
104	identifier not declared
105	sign not allowed
106	number expected
107	incompatible subrange types
108	file not allowed here
109	type must not be real
110	tagfield must be scalar or subrange
111	incompatible with tagfield type

112 index type must not be real
113 index type must be scalar or subrange
114 base type must not be real
115 base type must be scalar or subrange
116 error in type of standard procedure parameter
117 unsatisfied forward reference
118 forward reference type identifier in variable declaration
119 forward declared; repetition of parameter list not allowed
120 function result type must be scalar, subrange or pointer
121 file value parameter not allowed
122 forward declared function; repetition of result type not allowed
123 missing result type in function declaration
124 F-format for real only
125 error in type of standard function parameter
126 number of parameters does not agree with declaration
127 illegal parameter substitution
128 result type of parameter function does not agree with declaration
129 type conflict of operands
130 expression is not of set type
131 tests on equality allowed only
132 strict inclusion not allowed
133 file comparison not allowed
134 illegal type of operand
135 type of operand must be boolean
136 set element type must be scalar or subrange
137 set element types not compatible
138 type of variable is not array
139 index type is not compatible with declaration
140 type of variable is not record
141 type of variable must be file or pointer
142 illegal parameter substitution
143 illegal type of loop control variable
144 illegal type of expression
145 type conflict
146 assignment of files not allowed
147 label type incompatible with selecting expression
148 subrange bounds must be scalar
149 index type must not be integer
150 assignment to standard function is not allowed
151 assignment to formal function is now allowed
152 no such field in this record
153 type error in read
154 actual parameter must be a variable
155 control variable must not be declared on intermediate level
156 multidefined case label
157 too many cases in case statement
158 missing corresponding variant declaration
159 read or string tagfields not allowed
160 previous declaration was not forward
161 again forward declared
162 parameter size must be constant
163 missing variant in declaration
164 substitution of standard proc/func not allowed
165 multidefined label

166 multideclared label
167 undeclared label
168 undefined label
169 error in base set
170 value parameter expected
171 standard file was redeclared
172 undeclared external file
173 fortran procedure or function expected
174 pascal procedure or function expected
175 missing file "input" in program heading
176 missing file "output" in program heading
177 assignment to function identifier not allowed here
178 multidefined record variant
179 x-opt of actual proc/func does not match format declaration
180 control variable must not be formal
181 constant part of address out of range
201 error in real constant: digit expected
202 string constant must not exceed source line
203 integer constant exceeds range
204 8 or 9 in octal number
205 zero string not allowed
206 integer part of real constant exceeds range
250 too many nested scopes of identifiers
251 too many nested procedures and/or functions
252 too many forward references of procedure entries
253 procedure too long
In Pascal/Z, this error usually means that the compiler reached the last END and found no period (i.e., the BEGINs and ENDs do not match up).
254 too many long constants in this procedure
255 too many errors on this source line
256 too many external references
257 too many externals
258 too many local files
259 expression too complicated
260 too many exit labels

300 division by zero
301 no case provided for this value
302 index expression out of bounds
303 value to be assigned is out of bounds
304 element expression out of range
398 implementation restriction
More detailed explanations of the following implementation dependent error messages are available on pages 50 - 52 of the Pascal/Z Implementation Manual.

3980 Symbol table overflow
-3980 Type table overflow
3981 Function value may not be qualified
3982 Jump out of a procedure/function not allowed
3983 Non-string compared with string
3984 Program has too many levels of nesting
-3984 No more than forty fields in a record
3985 Cannot input/output this value because compiler option P was disabled when this enumeration type was declared.

```
  3986   Line or symbol too long
  3987   String too long
  3988   String too small for call by reference
 -3988   BCD number passed by reference must match exactly
  3989   EXTERNAL must be declared in main program

399  variable dimension arrays not implemented
```

There are a few other, unnumbered error messages which may be generated either during compilation or during run-time. These are described in detail on the page numbers given in parentheses after the brief description here.

Program too complex -- Usually means that there is not enough memory in the system to compile (page 51).

Too many errors -- Integer constant out of range of the allowable integer values (page 51).

Premature EOF -- Error encountered when using separate compilation. CP/M file names do not correspond to the internal module names OR fourth drive letter was not specified during compilation (pages 51, 71).

Stack overflow -- Stack space has been exhausted (page 51).

Unable to overlay -- Overlay module being called is not on currently logged-in drive (page 100).

Unable to chain -- PAS248 (or PAS254 if using 54K version) is not on currently logged-in drive (page 100).

Can't find <program>.REL -- When using COMPILE.SUB, this message means that no drive letter was specified in the command line (page 49).

APPENDIX SEVEN

PASCAL/BZ

This section of the Pascal/Z Implementation Manual describes the differences between the original Pascal/Z compiler and its business counterpart, Pascal/BZ.

Pascal/BZ is a version of the Pascal/Z compiler which has been changed to accommodate the business user. The Pascal/Z floating point routines are replaced in Pascal/BZ by BCD (binary-coded decimal) fixed point routines to allow for the greater precision and accuracy necessary for the business programmer. Pascal/BZ provides precision of up to thirty digits (fifteen to either side of the decimal point) under user control. Pascal/BZ fixed point numbers of different sizes can be mixed within the same program, permitting great flexibility in business applications programming.

Pascal/BZ is available only in a 54K version (that is, a 54K TPA is necessary to compile programs with Pascal/BZ).

The following pages detail specific changes made in the software package for Pascal/BZ, as well as the use of Pascal/BZ BCD numbers.

USING PASCAL/BZ FIXED POINT NUMBERS

* Declaring a BCD number

A BCD constant, type or variable declaration will be of the form:

```
CONST TAX = 13.5;
TYPE SALARY = BCD X:Y;
VAR RATE : BCD X:Y;
```

where BCD is a reserved word denoting a fixed point binary-coded decimal number. "X" indicates the number of digits to be placed to the left of the decimal point and "Y" indicates the number of places to the right. (Note that this specification is not needed when declaring a constant, since a constant will not change and cannot have a new value assigned to it.) There may be up to fifteen digits to either side of the decimal point.

A constant cannot be defined as the negative of another constant. For example:

```
CONST X = 14.7;
      Y = -X;
```

is not permitted, but

```
CONST X = 14.7;
      Y = -14.7;
```

is allowed.

BCD types and variables may be of varying sizes, and these do not have to match when performing arithmetic operations. The only exception to this rule is that when passing a variable by reference, the declarations of the variables must match exactly, or the error code -3988 will be generated by the compiler. When passing a BCD number by value, the number is converted to the size of the formal parameter (as declared), not the actual parameter (as passed at run-time).

Integers may be used in BCD expressions (provided of course that they are smaller than MAXINT or greater than -MAXINT). An integer used in a BCD expression will be converted by the compiler internally to a 15:15 BCD number, although its type and value will remain unchanged.

BCD numbers such as .1 or 1. may not be used. A zero must be placed before the decimal point in the first case (0.1) and after thedecimal point in the second case (1.0).

When performing BCD operations, the standard operators may be used (including relationals). The expression is evaluated, and the return value will be of size 15:15. This value is then automatically converted to the size of the variable to which it is finally assigned. Note that using the operator / (real divide) will yield a BCD number, regardless of the types of the operands.

If the number of digits to the left of the decimal point (excluding leading zeros) is less than or equal to that in the assignment variable, the entire process will take place with no problem. If the number of digits to the left is bigger than the size of the assignment variable, it will result in a run-time error.

If the number of digits to the right of the decimal point in the return value is greater than the number of digits in the assignment variable, the number will be truncated to fit.

If assigning a constant to a BCD number or using a constant in a BCD expression, the same rules apply.

* Files of BCD numbers

A file of BCD numbers is declared as follows:

```
VAR NUMFILE : FILE OF BCD X:Y;
```

The numbers will be written out to the file as the size which they are declared to be. For example, a 6:4 number will be written out in six bytes (see section on PASCAL/BZ STORAGE), and

a 5:7 number will be written as seven bytes. A file must be read as the same type in which it was written, that is a file written as BCD 6:4 must be read as BCD 6:4.

OUTPUTTING BCD NUMBERS

When outputting a BCD number, if no field width is specified, the variable will be written with a leading space (or a "-" sign if negative) followed by the number. The entire number will be right-justified with no leading zeros.

E.G.

WRITE(variable); where variable is 34562.12

will result in b34562.12

If a field width is specified, the variable will be written as described above, with a leading blank if positive, but will be right-justified in a field of the width indicated. If there are not enough places in the field to output the number as specified, the field width will be expanded to the right to accommodate the number. Leading zeros will be suppressed. Note that the decimal point occupies one place.

E.G.

WRITE(variable : 9); where variable is 34562.12

will result in b34562.12

WRITE(variable : 12); where variable is -34562.12

will result in bbb-34562.12

In both cases, the number will be printed as declared, with x digits to the left of the decimal point and y to the right.

FORMATTING BCD OUTPUT

Normally a BCD number may be output by doing a simple WRITE or WRITELN. Sometimes, however, it is desirable to format BCD numbers, and Pascal/BZ offers a variety of formatting options to accomplish this.

To format a BCD number, three types must be declared within the program, as follows:

```
TYPE BCD1515 = BCD 15:15;
     $STRING15 = STRING 15;
     $STRING40 = STRING 40;
```

Also, the FORMAT function must be declared as follows:

```
FUNCTION FORMAT( X:BCD1515; Y:$STRING15 ): $STRING40; EXTERNAL;
```

When calling the FORMAT routine to output a BCD number, several options can be specified in quotes, as shown below:

FORMAT(<BCD variable>, '<options>' : <field width>);

These options may be in any order, and can be separated by blanks. The options are:

$ Print the number with a leading dollar sign.
, Add commas every three digits to the left of the decimal point.
s Suppress trailing zeros.
m Print the minus sign after the number (the default is before the number for negative numbers).
p Print the negative number in parentheses rather than with a minus sign.
x Print x (where x is any integer between 0 and 15) digits to the right of the decimal point (the default is fifteen).
r Round the number to fit in x places (refers to above option; default is truncate).

Note that if any of the specified options conflict (for example specifying both m and p for a negative number), the result of the statement will be undefined.

The FORMAT function returns a string of forty characters ($STRING40), which can then be output using a WRITE or WRITELN statement.

The FORMAT routines are supplied in source so that they can be changed to accommodate differing format requirements. For European users, for example, the dollar sign can be changed to an appropriate monetary sign and the commas and decimal point can be reversed.

HOW TO USE PASCAL/BZ

To invoke Pascal/BZ, follow the instructions on HOW TO RUN PASCAL/Z (pages 46-49 of this manual), substituting PASCALBZ for PASCAL48 or PASCAL54.

When assembling, use BMAIN.SRC or BEMAIN.SRC (for separate compilation or EXTERNAL routines) rather than MAIN.SRC or EMAIN.SRC.

The library for Pascal/BZ is named BZLIB.REL, and is linked in automatically with the /E command when using Pascal/BZ (see section on HOW TO RUN PASCAL/Z on pages 46-49).

Using InterPEST with Pascal/BZ programs

If the debugger InterPEST (InterSystems Pascal Error Solving Tool) is to be used with a Pascal/BZ program, it must be compiled as described in the InterPEST Reference Manual, invoking PASCALBZ rather than PASCAL48 or PASCAL54. Then resulting .SRC file should then be assembled with BXMAIN.SRC or BXEMAIN.SRC (if using externals or separate compilation). The .REL file generated can then be linked with BZBUG.REL as described in the InterPEST Reference Manual.

BZBUG.REL contains a version of InterPEST designed to work with Pascal/BZ programs. In this version, all that applies to REAL numbers in the Pascal/Z version of InterPEST now applies to BCD numbers.

BCD numbers may be displayed, but may not be modified. When a BCD number is displayed, not only will the value be displayed, but also the declared size of the number will be given. The number will always be displayed as a 15:15 number, regardless of the declared size.

* Error messages

If the compiler generates any error messages referring to REAL numbers, they will in fact refer to BCD numbers.

The $F option to generate an error message if a floating point overflow/underflow is encountered will now refer to a BCD fixed point overflow/underflow.

* The Fixed Point Package

The Pascal/Z Fixed Point Package is not supplied as a part of Pascal/BZ, since the compiler has intrinsic BCD fixed point numbers. (The Fixed Point Package also requires REALs to perform many of its operations, and REALs have been removed from Pascal/BZ.)

* Pascal/BZ EXTERNAL routines

When using EXTERNAL routines as described on pages 74-78, BCD numbers returned by functions will be stored above the function parameters on the stack (see page 74). The least significant byte will be stored at n+8(IX), and the number will occupy space on the stack according to its size (as described in PASCAL/BZ STORAGE).

* TRUNC & ROUND

The Pascal procedures TRUNC and ROUND will work slightly differently with BCD numbers than they do with REALs. The numbers to be truncated or rounded will be converted to integers, and an overflow/underflow error will be generated if attempting

to truncate or round numbers which convert to greater than MAXINT or less than -MAXINT.

* PASCAL FUNCTIONS

In Pascal/BZ, none of the standard functions may be passed a BCD number, with the exception of ABS, SQR and SQRT. (See the section on PASCAL STANDARD FUNCTIONS for more information on these routines.)

PASCAL/BZ STORAGE

In Pascal/BZ, fixed point numbers are implemented in binary-coded decimal. The numbers may be variable in length, and are stored two digits per byte, organized as follows:

```
   byte    1    byte    2    byte    n
|sign/digit| |digit/digit| ... |digit/digit|.
```

(for digits to the left of the decimal point)

where n may be up to eight bytes to accomodate a number with fifteen digits to the left of the decimal point.

```
|digit/digit| ... |digit/digit|
```

(for digits to the right of the decimal point)

Thus for a number with an odd number of digits to the left and an even number of digits to the right, there will never be any wasted storage. For a number with an even number of digits to the left or an odd number of digits to the right, there will be some waste, as shown below:

For a 3:4 BCD number > 2 bytes left, 2 bytes right = 4 bytes (no waste)
For a 2:4 BCD number > 2 bytes left, 2 bytes right = 4 bytes (one wasted nibble to the left)
For a 1:4 BCD number > 1 byte left, 2 bytes right = 3 bytes (no waste)
For a 4:5 BCD number > 3 bytes left, 3 bytes right = 6 bytes (one wasted nibble to the left, one wasted nibble to the right)

(Note that an even:odd number is the most inefficient, with a total of two wasted nibbles.)

If there is wasted space on the left side of the decimal point the wasted nibble will be the nibble after the sign; on the right side of the decimal point the wasted nibble will be the nibble after the last digit.

The maximum precision allowed in Pascal/BZ is fifteen digits to

either side of the decimal point, for a total of thirty digits. See USING PASCAL/BZ FIXED POINT NUMBERS for more information on how to make use of BCD numbers.

PASCAL/Z COMMENTS & BUG REPORTS

Please use this sheet (and any additional sheets if necessary) when sending your comments and bug reports to INTERSYSTEMS.

NAME: _____

ADDRESS: _____

_____

_____

PHONE: _____

VERSION: _____ SERIAL NUMBER: _____

BUGS (include a listing, description of the problem, etc):

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

SUGGESTIONS FOR IMPROVEMENTS TO THE COMPILER:

_____

_____

_____

_____

_____

_____

_____

SUGGESTIONS FOR IMPROVEMENTS TO THE MANUAL:

EXTENSIONS:

WHAT DO YOU THINK OF EXISTING PASCAL/Z EXTENSIONS:

WHAT EXTENSIONS WOULD YOU LIKE TO SEE:

DO YOU HAVE ANY UNANSWERED QUESTIONS ABOUT PASCAL/Z:

INDEX

abs 31,119
absolute value 10,31
ABSSQR.SRC 10
access 6,11,36,43,58,62
accumulator 74,75,76
activation 76,93
activation records 76,93
actual parameter 115
ADD 101,102,104-107
add 10,11,36,101,102,104-107
address 74,75,76,77,87,97-99
ADDSUB.SRC 10
allocation 10,36,79-84,87-88,95
and 37
append 11,64,66,68
appendices
  one 97-99
  two 100
  three 101-107
  four 108
  five 109
  six 110-113
  seven 114-120
ARCTAN.SRC 10,31
arctangent 10,31
array 15,18,22,37,39,40,43,53-54,86,93,98
ascii 16,21,37
ASMBL.COM 9,13,48
asmble 2,9,48
assembler 2,4,9,13,47-48,70-73,74-78,79-80,92
assignments 20,25,39,42,43,64,114-115

base types 93,94,97
BCD 10,35,51,101,114-120
begin 28,37
BEMAIN.SRC 117
binary 89
binary-coded decimal 10,35,51,101,114-120
bit 11,16,38,89,99
block 5,30,32,35,54
BMAIN.SRC 117
boolean 16,25,26,27,34,39,53,54,59,62,74,90,93,
97-98,102
bound 42,93
brackets 17,37
buffer 10-11,38,54,97
byte 34,36,38,39,40,43,54,67,69,74-77,89,92,93-94,
97-99,101,115-116,119-120
BYTIN.SRC 10
BYTOT.SRC 10
BXEMAIN.SRC 118
BXMAIN.SRC 118
BZ 35,51,114-120
BZBUG.REL 118
BZLIB.REL 117

CALL 74,92
call 10,25,40,51,54,74,79,92,93,96,97,101
call by reference 51,77,97
called 30,48,74,76,101
case 4,5,27,37,39,41,43,61,69,93
case label list 27,39,61
case selector 27,69
CHAIN.SRC 10,83,86
chain 10,34,80,83,86,100
char 15,16,22,31,53,69,86,92,97
character 10,11,15,16,31,37,38,51,67,70,85,92,93
check 10,15,36,41-42,43,67,69,70,95,86,100
CHKD.SRC 11
chr 31
CLSOT.SRC 11,96
CMAIN.SRC 11,86
CMPCHK.SRC 10
.COM 48
comma 102-103,117
command tail 54
comments 4,17,37,41,85
compare 25,103
compatibility 40
compatible 6,7,15,25
compilation 46-49,79-84
COMPILE.SUB 9,49
compiler 2,3,4,34,36
complement 10,89
compound statement 28,29
CON: 60
concatenate 66
conditional statement 27
CONSOL.SRC 11
console 10,11,38,46,47,53-54,59,60
console input 60
console input buffer 38,54
console output 60
const 4,22,32,37,63,104,114-115
constant 4,9,14,15,16,22,27,32,36,43,51,61,63,93,
94,101-107,114-115
constant folding 36
constant strings 67
construct 28
control C 41,47,96
cos 31
cosine 11,31
cpi 77
CVTFLT.SRC 11
CVTSFP.SRC 11

data 14,15,16,17,18
data areas 43,86
data types 16-21,90
dcx 77
deallocation 10,20-21,87-88
DEBUG.REL 9
debugger 4,9,48
decimal 9,89,90-91,101-104,114-120
decimal point 90-91,114-120
declaration 32,34,40,44-45,47,50,51,66,70,71,86,93,101-107,
114-115,116

declaration level 75-76
decrements 36
DECS 9,34,46,49,100
default 10,39,41-42,46,71,85
define 22,31
definitions 9,11,14,15,16,75
DEFLT.SRC 11
delete 10,53,59
density 100
device 10,53,60
device input 60
device output 60
digit 16,37,38,71,89,90-91,101-104,114-120
Digital Research 1,2,6
direct file access 4,6,11,58,62
directory 10,59
disable 41
dispose 5,40,87
div 37,63
SDIVD 101,102
DIVD.SRC 11
divide 10,11,36,42,101,102,115
divide by zero 36
dollar sign 41,85,102-103,117
DONE2.SRC 11
downto 26,37
DSKFIL.SRC 11
DYNALL.SRC 11
dynamic storage 5,10,20-21,87-88,93-94,95

editors 37
efficiency 3,22,29,43,44-45,66,69,93
element 18-19,23,25,38,43,64,93,101
element type 18-19
EMAIN.SRC 9,71,73,74-78,82-83
enable 41-42
end 10,28,32-33,37
end of file See EOF,EOFLN
end of line See EOLN,EOFLN
enter 11
ENTEXT.SRC 11,96
entr 75-78,86
entry 42,69,70-71,75-76,92
entry point 70,75-78,79,83,92
enumeration type 4,16,17,42,51,53-54,62,93,97,103
eof 31,51,53-54,58
EOFLN.SRC 11
eoln 31,53-54
equivalence 44-45
ERROR.SRC 11
error messages 11,41,42,50-51,70,71,100,115,118
errors 7,10-11,17,34,38,39,40,41-42,47,49,50-52,54,
67,70,71,89,92,110-113,115

EXAMPLE 10
execution 41,42,48-49,76,79,83,95
execution speed 34,43
exit 10,26,48,54,75-77
exp 31
EXPFCT.SRC 11
exponent 75,89,99

exponential notation 10,91
expressions 25,26,30,36,39,61,115
ext 71
EXTENS 10
extensions 3,4,9,46-47,61-62,63-85
external routines 4,9,37,38,51,61,66,70-73,74-78,79-84,97,118
EXTD 78
EXTR 78

FADDSB.SRC 11
FCTMAC.SRC 11
fields 23,38,51,64
field width 90-91,116-117
FILE 15,16,19,31,37,62,97
file 4,6,10-11,15,16,19,21,31,34,37,38,39,53-54,71,
79-84,86,87,115-116
file data 53
FILEIO.PAS 10,56
file names 46-47,51,53,54,59,70,71,79
file variables 87
FILEXT.SRC 11
FILNAM.SRC 11
fini 96
FIXCONST.PAS 9,101-107
FIXED.PAS 9,101-107
fixederror 102
fixed point arith. 9-11,35,89,91,101-107,114-120
FIXEDEX.PAS 10,104-107
fixtoreal 102
fixtostr 102-107
FIXTYPE.PAS 10,101-107
FIXVAR.PAS 10,101-107
flag 102
floating point 10-12,16,35,41,89,90-91,96,99,114
floating pt. formats 89,99
floating point output 90-91
FLTIN.SRC 11,96
FMULT.SRC 11
FOR 21,26,37,39,41,94
FOR loop 26,39,41,94
formal parameter 115
FORMAT 116-117
format mode 103
formatting output 90-91,116-117
forward declarations 92
foster, charles 108
FOUT.SRC 11
FPDIVD.SRC 11
FPERR.SRC 11
FPINIT.SRC 11
FPMAC.SRC 11
FPRLOP.SRC 11
FPSQR.SRC 11
FPTEN.SRC 11
ftxtin 86
function 4,5,10-11,25,30,31,37,38,39,40,42,47,50,51,
53,54,59,61,64-65,66-68,71,74,75-78,79,92,
93,101-104,116-117,118,119
function activation 93
FXDCVT.SRC 11

get 5,40,53
global data areas 43,86
global declarations 51,86
global level 23,43,70,75
global variables 23,95,101,102
GOTO 5,14,39,41

hardware problems 100
heap 86,87,95
HELLO 9,13
hex 21,48
HOWTO.RUN 9

I/O ports 76-77
identifier 32,34,37,38,39,63,101
imbed 41
implementation 4,14,38-40,50,61-62
include files 4,41,61,85,103-107
increments 36
INDEX 66-67
index 18,42,43,55
INDIR.SRC 11
INFO.NEW 9
initialization 11,34,95
INPT.SRC 11,96
INPUT 39
input 11,34,38,39,40,42,51,53-54,60,61-62,77
input files 10-11,39,53-57
installing in ROM 96
integers 4,10-11,15,16,31,34,38,42,43,51,53,63,90,93,
94,97,115
interactive 4
intersection 11,19
intrinsic data types 16
intrinsic procedures 66,87
introduction 2,14

Jensen & Wirth 2,12,20,32,38,40,90
jmp 92,96
jump 39,43,50
jump table 39,69

keyboard 21

L0.SRC 11
label 27,32,37,78,92
language 2,4,14,15,34,38,61,66,70,74-77
LAST.SRC 11,86,95
legality 16,28,32,53,64-65,67,70
LENGTH 11,64,66-68,99
length 38,51,61,64,66-68
LIB.REL 9,10,48,59,79-80,83,103
library 2,9,10-11,48,66,70,79-80,83,86,103
library modules 9-11,48,79-84,86,95,96
limitations 40
line 10,31,32,38,51,53-54
LINK.COM 9,13,49
linked lists 20-21,87
linker/loader 2,9,40,48-49,59,70,75-78,79-84,86,92
list 43,69,74

listing files 9,34,41-42,46-47,50
load 10,11
local data areas 30,43,76
local level 23,30,43
local variables 23,30,40,75,93
log 11
logarithm 31
LOOK.SRC 11
loops 26,33,39,41,94
.lst 9,34,46-49,50-52
.LST: 60
LVL 75

macro-code 2,4,6,34,41,46-47,74-78
macros 75-78,86,96
MAIN.SRC 9,10,11,38,47,48,54,70,74,95,96
MAINMAP 79
main module 70-71,79-84
main program 40,51,70,79-84,85,86
mantissa 11,89,99
MARK 5,40,87-88
maximums 16,38-39,51,54,66-67,69
MAXINT 16,38,115,119
maxout 38,54
membership 11,19
memory 6,10,40,51,79-80,93-94,95,97,100
memory locations 10,11,79-84,95,96,97
memory requirements 6,40,51,93,101
memory usage 40,93-94,95,97-99
minuend 102
mod 37
MPNORM.SRC 11
$MULT 101,102
MULT.SRC 11
multiply 11,42,101,102

named equivalence 44-45
NATLOG.SRC 11
negative numbers 90
nesting 30,38,39,51,85
nesting levels 34,38,50,51
NEW 5,20-21,40,87-88
nibble 119
non-reals 36,39
null 33,66
numbers 10-11,16,31,37,38,43,63,89,90,91,99,
101-107

object code 2,9,47,96
odd 31
open files 38,53-54,96
operands 15,103
operations 15,40,61,66,102-103,114-115
operators 4,115
OPFILE.SRC 11
optimization 4,5,36,43,96
optimizer 10,47-48
options 41-42,46,48,50,51,69,71,85,116-117,118
ord 31
ordinal 31,97
ordinal value 69

.ORG 47
out of range 42
OUTPT.SRC 11
OUTPUT 39
output 42,47,51,60,61-62,76,90-91,100,116-117
output files 6,10-11,34,38,39,41,46,53-57,87,96
overflow 42,50,51,95,102,118
overhead 93
OVERLAY.SRC 10
overlays 4,9,10,34,61,79-84,86,100
overlay modules 9,34,46,79-84,100
OVLYGEN.COM 10,79-80,83
OVLYMAP 79,83

pack 39
padding 90
page 5,40
pairing 95
parameters 30,38,40,42,43,51,53,54,65,67,70,74-77,92,93,
114-115,118
94,97-99,103
parentheses 25,37
parity 37
parmsz 76
.PAS 34,46
PAS248 9,34,46,49,100
PAS254 9,34,46,49
PASCAL48 9,34,46,49,100
PASCAL54 9,34,46,49,100
PASOPT 10,47-48
passing by value 30,65,67,102,115
passing by reference 30,43,51,67
PEEK 10
PFSTAT 9,34,46,49,100
pointers 20-21,23,40,87-88,93,99
ports 76-77
precision 9,11,16,38,89,101-104,114-120
pred 31
PRIMES 10
print 10
printer 46-47,60
procedural parameters 5,40
procedure 5,10,20,30,31,37,38,39,40,42,47,50,51,54,
66-67,70,71,74-78,79,87,92,93,101-104
procedure activation 93
processor 4,36
program structure 32
PSTAT.SRC 11
PUT 5,40,53

query option 80,83
quoted strings 37,54,67,79,86,93,94

random access 4,6,11,58,62
range-checking 10,42,43,67,69
ranges 17,38,93
RBLOCK.SRC 11
read 10-11,21,40,53-54,58,60,92
readln 11,53-54
real 15,16,26,31,34,38,41,43,53-54,74,75,90,93,99,
101,102,114,118

realtofix 102
record 15,18-19,20-21,23,29,37,38,39,40,43,51,54,58,
62,64,76,93,98
record numbers 58
record zero 58
recursion 3,30,52
re-entrancy 3,4,95
reference parameters 94,97
references 22
registers 10,11,36,74-77,92,95
relational operators 11,25,67,103,115
relative addresses 95
relative jumps 43
relocatable modules 9,48
relocatable object
code modules 48
relops See relational operators
RENERA.SRC 10,59
RENDRV.PAS 10,59
repeat 26,37,39,41
repeat loop 26,39,41
reserved words 37,114
reset 12,53,58,60
RESET.SRC 12
resident program 79
restrictions 5
return values 53,74-77,115,118
rewrite 12,53,60
REWRIT.SRC 12
ROM 3,4,95,96
ROTATE.SRC 12
round 12,31,89,117,118-119
ROUND.SRC 12
roundoff error 89

S-101 bus 100
SAVREG.SRC 12
scalars 15,17,22,26,30,31,39,64,74
scientific notation 91
search option 80,83
semi-colons 32-33
separate compilation 4,9,51,61,70-73,77
separate modules 51,61,70-73,79-84
separators 32,37
sequential access 58
sets 11,15,19,37,38,43,94,99
SETCON.SRC 12
SETFTN.SRC 12
setlength 11,66-68
shiftleft 101
shiftright 101
sign 63,89,90,99,101-107
significance 38,70,75,103
sin 30,31
SINCOS.SRC 12
sine 11,31
size 11,21,34,38,40,44,54,69,80,86,96,102,114,115,
119-120
source code 2,4,6,10,41,46,59,86,101
spaces 37,41,46,85,90

specifications 38-40
sqr 31,119
sqrt 12,30,31,119
SQRT.SRC 12
square 10,11,31
square root 11,31
.SRC 9,10-11,34,46-49,77-78
SRELOP.SRC 12
stack 11,34,42,51,74-77,85,93-94,95,97-99,118
stack overflow 42,51
standard functions 31
standard language 4,5,14,17,31,37,40,53,66
standard procedures 31
statements 4,14,20-21,25,26,27,28,29,30,32,33,39,41,42,
storage 5,10,20-21,23,36,39,40,75,76,87-88,93-94,103
STRFCT.SRC 12
string constants 22,43,63,86,93,94
strings 4,10-11,22,37,38,43,51,53,61,63,66-68,79,86,,
93,94,99,102-103
strlop 11
strtofix 102
structured
equivalence 44-45
SUB 101,102,104-107
subbyte 101
submit 9,49
subranges 17,43,93
subroutines 30,48
subsets 19
substrings 66
subtract 11,36,101,102,104-107
succ 31
supltzer 103
suplzer 103
InterPEST
(InterSystems Pascal
Error Solving Tool) 4,9,48,118
.SYM 70
symbol 21,32,34,37,38,51,52,70-71,83,92
symbolic input/output 4,42 Also see enumeration types
symbol table 6,34-35,50
syntax 32,58,71
syntax errors 50

tabs 37,52
tables 69
TEST.SUB 83
text 12,16,31,53-56,60
TEXT.SRC 12
text files 40,46,61,92
threaded code 3
trace 11,42,50,71
transportability 61
trigonometric fcts 31
troubleshooting 100
trunc 31
truncate 11,31,115,117,118-119
.TYP 71
type 10,15,16,17,18,19,20-21,23,26,30,31,32,34-35,
37,40,42,44-45,50,53,54,61-62,65,66-67,70-71,
90,94,97,101-107,114,116

type declarations 15-21,32,34,40,44-45,50
type table 6,34-35,44-45,50,67

U.C.S.D. 9
UCTRANS 10
underflow 41,118
union 11,19
unpack 39
unstructured relops 11
until 26,37
updates 7
URELOP.SRC 12

value parameters 42,94,97-99
values 30,31,39,50,51,61,62,64,67,89,95,103
var 23,37,94,104,114
variable 10,15,16,17,20-21,23-24,25,26,27,30,32,36,39,40,
43,45,53,58,60,65,74-76,93,95,97-99,101,103,
114-116
variable access 24,36,43
variable declarations 23-24,32,40
variable length
strings 4,10-11,61,66-68,99
variable storage 20-21,36,87,97-99
variant records 14,87
verbose option 80,83
vocabulary 37
vsiz 75

warranty 109
wboth 103,104-107
wcomma 103
wdollar 103
while 26,37,39,41,
while loop 26,33,39,41
Wirth 2,43
with 29,37,39,43
words 37
write 11,40,53-57,58,60,90-91,92
writeln 11,54-57

XEMAIN.SRC 9
XMAIN.SRC 9

zero 10,11,36,39,66,74,99,102,103,115,117